Availability and performance aspects for mainframe consolidated servers

Klaus Johansen
s053075

Kongens Lyngby 2007

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 4525 3351, Fax +45 4588 2673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-Thesis-2007-94

ABSTRACT

Most people believe that the mainframe platform currently suffers a slow but a certain death. This is however a common misconception: The mainframe is very much alive and going strong.

This thesis recognizes the mainframe platform's superior qualities, and deals with one of the platform's new driving forces: Server consolidation based on the z/VM operating system and Linux.

Some effort is first made to provide a basic understanding of the IBM mainframe architecture in order to establish common ground. A concise, yet theoretically and practically balanced, description of z/VM based virtualization follows. This provides amongst other things insight into how resources (processors resources in particular) can be shared.

The report also introduces some of the most important issues, which set apart mainframe Linux from Linux on other platforms. Performance tests have been carried out, and through these an insight into the systems behaviour can be gained.

The thesis also includes initial considerations on monitoring software for the virtual server environment on the mainframe.

CONTENT

PREFACE

The thesis at hand is written by the undersigned in order to fulfil the last requirements for the M.Sc. degree in Computer Science Engineering at the Technical University of Denmark (DTU). The work constitutes 30 ETCS points.

The project was carried out in the period from March to the end of September 2007. The thesis project has been supervised by Associate Professor Hans Henrik Løvengreen from the institute of "Informatics and Mathematic Modelling" (IMM) at DTU.

The thesis is written in industrial collaboration with "KMD A/S", Lautrupparken 40-42, DK-2750 Ballerup, Denmark. More specifically the project has been coupled with the department of "Technical Basic operation Mainframe" (TBM) lead by Peter Aksel Mortensen (PTM). KMD has kindly provided office space, workstation, and a z/VM - Linux test environment for the project.

Acknowledgements

I would like to thank my supervisor Hans Henrik Løvengreen: First of all for him accepting this project in the first place considering the topic generally lies outside IMM's main research areas. Next for his good advice regarding form and structure of this paper; and finally for his calm approach, patience, general encouragements, and his sense of details.

I would also like to thank the people at KMD: Peter for "letting me in"; Bo, Frank and Torben for answering my questions and valuing my opinion; and Thomas and Mikkel for making the working days a little more fun e.g. by endlessly comparing the mainframe with a coal-fired steam-engine. And last but not least I would like to thank Kristian for proof reading most of the report.

Klaus Johansen

2007-09-28

1. INTRODUCTION

1.1 Background and motivation

1.1.1 Mainframes are flourishing

The mainframe has been declared dead many times over the years; and the word itself reminds most people of an old-fashioned colossus with large spinning magnetic tapes and a typewriter like interface. It is common belief that mainframes only exist to support old applications, which nobody can effort, or knows how to rewrite. Nothing could be more wrong.

The mainframe has evolved into the most secure, reliable, and capable computer platform available. It is correct that it still run many of the same workloads/applications as it did decades ago, but today it also impresses in running new workloads, e.g. in the world of J2EE (Java Enterprise Edition) and SOA (Service-Oriented Architecture).

The mainframe sales confirm that the technology is “alive and kicking”. Michael Loughridge, IBM’s senior vice president and chief financial officer, is quoted for saying, that IBM with second-quarter 2007 had “eight consecutive quarters of growth in the mainframe business” [73]. More over [54] claims that IBM has confirmed that the System z [mainframe] capacity shipped in fourth-quarter 2006 was greater than the total capacity of the then current installed IBM mainframe worldwide inventory.

Server consolidation is one of the new driving forces for the platform. The Linux ports for IBM S/390 and IBM zSeries combined with z/VM virtualization makes it possible to run hundreds of individual Linux servers on the same mainframe box. High hardware and licensing costs are compensated by 90% Linux workload price reductions (IFL engines), savings in power consumption, data center floor space, administration costs, etc.

IBM has recently announced (Aug. 1, 2007) that they themselves plan to consolidate about 3,900 computer servers onto about 30 System z mainframes running Linux. IBM estimates that the new server environment will consume approximately 80 percent less energy and they expects significant savings over five years in energy, software and system support costs. [38].

The virtualization and consolidation idea is completely in line with the increased focus on server virtualization on other platforms within the latest years: Intel and AMD have implemented virtualization support in their x86 processors; and virtual server solution (e.g. based on Xen and VMware) are getting more and more prevalent. The mainframe however seem to be far ahead: It has supported virtualization in 35 years and the environment is correspondingly mature; the whole hardware architecture is optimized for virtualization.

1.1.2 The KMD server consolidation case

1.1.2.1 The Company

KMD A/S is the largest IT company on Danish hands, with a turnover of more than DKK 3 billion. The total share capital of KMD A/S is held by the parent company "Kommune Holding A/S", which in turn is owned by the National Association of Local Authorities (KL). [39]

KMD provides IT and consultancy services to the public and private markets. The core business is, and has historically always been, products for the Danish local authorities (the municipalities). The corporate market, however, is an important part of KMD's growth strategy. Today KMD delivers maintenance and development projects to e.g. Q8, HK, and ATP. KMD is also engaged in operations outsourcing for companies such as Coop Danmark, Jysk Nordic and uni-chains. [39]

In order to provide the IT services, KMD operates around 3,000 servers (running Microsoft Windows, Linux, and a few UNIX variants). The company also has two IBM System z9 EC, model S54, mainframes; each having about half of the 54 available processors activated (the remaining are available for "capacity on demand" upgrades). The two mainframes are placed in separate data centres, with mirrored storage. They mainframes run many "traditional" and core business (especially transactions based) workloads.

1.1.2.2 Perspektiv ASP - Server Consolidation

As a small piece in the big puzzle, KMD develops and sells a payroll application suite, "Perspektiv" (in English: "Perspective"), to corporate businesses. The Perspektiv customers include: the Danish Broadcasting Corporation (DR), F.L. Smidth & Co A/S, TDC Services A/S, and F-Group. The application runs on most UNIX platforms, including Linux, HP-UX, AIX, Tru64, even VMS. Several backend database management systems (DBMSs) are supported: Informix, Ingress, and Oracle.

Perspektiv is also offered as an ASP (Application Service Provider) solution, where KMD supplies, houses, and maintains the server infrastructure needed to run the application. Capacity exhausting of the existing ASP servers has triggered migration of the application from the existing HP/HP-UX platform to Linux under z/VM on the mainframe. IBM Total Cost of Ownership (TCO) calculations have predicted significant savings compared to an adequate HP based solution.

The mainframe benefits from the hardware already being in house; and its extreme scalability and "capacity on demand" offers, which basically makes an upgrade as easy as turning a knob; this without occupying further data centre space, consuming extra electricity or producing more heat. Perspektiv furthermore inherits the honoured reliability of the mainframe hardware, which is redundant in thinkable way and autonomously calls IBM for replacement parts on failover.

1.2 Purpose and goals

In the academic world the mainframe, and especially mainframe virtualization technology, has been generally overlooked for years. The topic is sparsely researched and the platform is largely ignored from an educational point of view. This is a natural effect of the general perception that the mainframe is dead and a bare memory of the past. As mentioned above, the reality is quite different, and there is absolutely no indications that the situation is about to change.

KMD has already started the server consolidation process: A handful of Perspektiv ASP customers are now running Linux on the mainframe. KMD successfully operates the new z/VM-Linux environment, and with very promising results. It is nevertheless still a young platform from KMD's perspective.

KMD wishes a deeper understanding of the new environment, e.g. of the interaction between the hardware, the virtualization layer (z/VM), and Linux. The ability to control and predict distribution of resources (especially CPU resources) is also of interest. Unfortunately the existing literature tend to be very "how-to oriented", unnecessary detailed, or unfeasibly extensive (IBM's "z/VM library" alone include over 50 books and ten thousands of pages).

This thesis is intended to help KMD (and possibly others) to achieve a better understanding of the new environment. In the light of the lacking academic interest for mainframes and the missing knowledge about them, this thesis is expected to provide an introduction to mainframe architecture as well. To sum up, the report is expected to:

- Provide an introduction to the mainframe architecture.
- In details describe the virtual environment and thereby explain the interaction between, and the mutual support of:
  - the hardware and hardware partitioning
  - software virtualization layer (z/VM)
  - Linux as operating system in a virtual machine
- Show the resource sharing / virtualization options, and in particular the possibility to control and predict processor resource distribution.

Finally, the gained platform insight should be used to address the matter of availability: "monitoring" in particular. It is KMD's goal to ensure optimum and flawless 24x7 operation of the mainframe based Perspektiv ASP solution. Similar to all other platforms within KMD, a monitoring solution is expected to help accomplish this goal. This project in particular should:

- Determine which components and layers need monitoring; thereby state monitoring software requirement.
- If possible find and integrate monitoring such software.

1.3 Methods

The mainframe is presented from a purely descriptive point of view, by introducing its history, the main elements of the hardware architecture, and the most used mainframe operating systems.

The chapters describing z/VM and Linux use the same approach. The z/VM chapter is however supported by more theoretical sections e.g. providing a basic on virtualization theory. The z/VM and Linux chapters also have a practical angle, giving concretely examples on configuration and commands.

Generally the understanding of the system has been obtained empirically. In the report, this is especially is reflected by the performance test chapter, and the in Appendix given test tools and scripts, which have been programmed to obtain the empiric data. The test results themselves and several other spin-off findings are (or have been) directly applicable for KMD. A method developed to calculate processor resources distribution (section 3.6.1.1) is also to a large extend empirically founded.

1.4 Reader's Guide

Report organisation

The thesis report is divided in seven chapters; the first of which (the introduction currently at hand) presents the background and objectives for thesis project; plus practical issues regarding the report structure.

Chapter 2 gives a brief introduction to the IBM mainframe. The chapter presents important terms from the distinctive mainframe terminology and provides a basic understanding of the mainframe hardware and architecture.

Having established common ground, **chapter 3** focuses on the z/VM operating system and its component. It deals amongst other things with the virtualization concept and the ability to share/distribute system resources. **Chapter 4** is somewhat related to the preceding chapter: It introduces Linux as mainframe operating system; how it differs from Linux on other platforms; and special virtualization related issues.

Chapter 5 includes description and results of four performance tests. Besides concrete optimization recommendations, the chapter should contribute to a better understanding "mainframe Linux". Be aware the main section numbers match the test numbers, which are used in appendixes, etc.

Finally **chapter 6** deals with the concept of availability or "monitoring" in particular. It presents initial considerations in relation to finding adequate monitoring software for KMD's mainframe Linux environment. .

The **conclusion,** given in the end of the report (p. 95), gives a brief roundup of project achievements.

Appendixes

Two appendixes are provided: **Appendix A** provides supplementary information regarding the performance tests from Chapter 5. **Appendix B** is similar related to the performance test: it contains source code for most of the developed test programs and scripts.

Index and glossary

The index in the end of the report (page 102) provides a method to find the page where a particular concept, keyword, technology, acronym or similar is explained or mentioned. The words are typically written in boldface to make them easier to distinguish in the text. A real glossary is not provided being an almost impossible task to write. IBM, however, provides a z/VM specific glossary [8], which can be downloaded here:

http://publibz.boulder.ibm.com/epubs/pdf/hcsl9b00.pdf

Bibliographic references

Bibliographic references are indicated with square backets [ ] and the number refer to the bibliography section on page 92. References to particular page, #, within a source are given in the brackets after a comma: [ref, p. #]. Main sources, which form the basis for whole or multiple sections are introduced in advance or are listed in the end of the relevant section in a separate paragraph/on a new line (see below).

Single or specific statements/facts are referenced within the sentence before full stop [like his]. Sources related to a complete paragraph/section are mentioned after full stop. [like this][and this]

[Section reference 1] [Section reference 2]

Vendors and definition of the term "mainframe"

Throughout this paper the word "mainframe" solely refers to IBM mainframe products and if nothing else mentioned "IBM System z9 Enterprise Class (EC)" – the current top model. Other products like ClearPath from Unisys, Nova from Fujitsu, NonStop from Hewlett-Packard exist but IBM dominates the market (market share estimated above 90% [24]) and finally the mainframes at KMD are System z9 EC.

Software versions

The IBM documentation library used for this project is mainly related to z/VM V5R2. All test and experiments are similar conducted on SUSE Linux Enterprise Server version 9 (SLES) running under z/VM V5R2 (the software versions available at KMD). Actually z/VM V5R3 was released halfway though the project, and SUSE Enterprise Server version 10 (SLES10) has been available for some time. However, none of the principles or examples mentioned should be affected by the new versions (unless otherwise stated).

2. The IBM Mainframe

This chapter gives a brief and very basic introduction to IBM mainframe architecture and hardware platform. Readers which are familiar with the IBM mainframe platform are encouraged to skip to chapter 3 “IBM z/VM” on page 19. The introduction is not in any way exhaustive but included to establish a minimum of “common ground”. *Parts* of the text is *revised* sections from chapter 2 in “Execution and monitoring of Linux under z/VM” [62] – a preceding paper written to assess the prospects of the thesis at hand.

2.1 Mainframe History

The IBM mainframe or the “Big Iron” (industry jargon) originates from IBM System/360, which was introduced in 1964. The System/360 was the first general purpose computer running *both* commercial and scientific application (same standard hardware, different programs). A considerable number of programs from the 1960s and 1970s are still used today proving one of the mainframe core qualities: extreme **backward compatibility**.

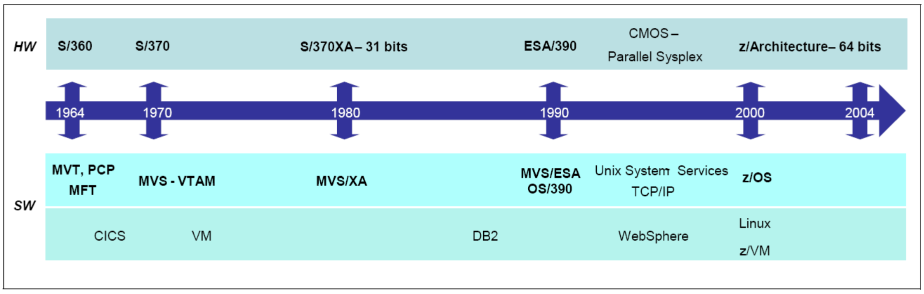


Figure: 2-1 IBM Mainframe time line [57, App. A]

Roughly every ten years (see Figure: 2-1) IBM has significantly extended the platform. System/370 introduced **multiprocessor** capabilities in 1970, allowing more than processor in the same system and sharing memory. System/370 was also the first lines of computers using **virtual memory**. The System/370 Extended Architecture (S/370-XA) from 1982 extended the address space from 24 bit to 31 bit.

Around 1990 the success of PCs and small servers forced IBM to reinvent the mainframe from the inside. With System/390 IBM infused a new technology core and reduced prices. New concepts like Parallel Sysplex were presented offering system **clustering** and **automated fail over** across multiple system and thereby higher availability. CMOS-based processors had replaced the prior bipolar technology and reduced the physical size and power consumption radically.

The zSeries came in the year 2000 with z/Architecture supporting **64-bit addressing**. Specialized cryptographic capabilities were introduced and the ability to "downgrade" processors by microcode for specific workloads (Linux, Java) drastically reduced software costs. Among many other improvements the System z9-109 (July 2005) came with new instructions to improve virtualization overhead, further processors and yet extended I/O capabilities.

The current top model "System z9 EC" (Enterprise Class) is basically the z9-109, which has been renamed. This has been done to make the system easier distinguishable from a new mainframe family member: The System z9 Business Class (BC). The z9 BC offers is a midrange mainframe for small to medium sized enterprises. It offers a lower-capacity entry point and more granular growth options than z9 EC. [37]

[57] [63]

2.2 Hardware and Architecture

Mainframe terminology differs from general computer terminology in many ways. This chapter introduces the most important differences giving a brief introduction to the mainframe hardware and system architecture. The text is based on "Introduction to the New Mainframe: z/OS Basics" [57], unless otherwise mentioned.

2.2.1 The mainframe box - CEC

A complete mainframe box (as depictured in Figure: 2-2) is by many simply called a "system", but also very confusingly referred to as "**processor**", a "**CPU**", or a "**CEC**" (Central Electronic Complex).

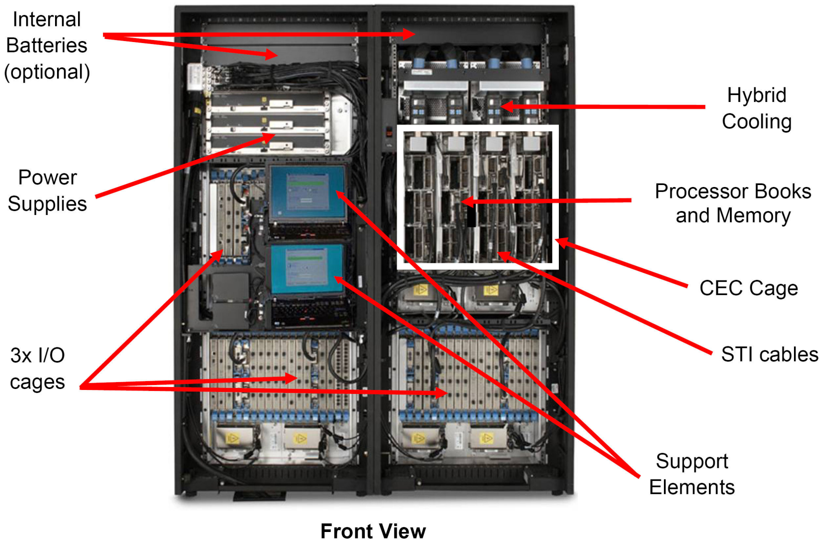


Figure: 2-2 Inside IBM z9-109 (S38 or S54) / System z9 EC. [20] The box measures 1,94 x 1,57 x 1,57m (H x W x D) including covers, and thereby occupy 2,49m^2.

2.2.2 Processors and computation

The IBM System z9 mainframes operate with several types of processors/CPUs. Physically there are all alike, but IBM configures or "characterizes" them for a specific purpose.

Processor units: PUs

The un-characterized processors are called PUs (Processor Units). PUs are basically physically available but not enabled processors, which function as spares. These spares can replace failing processors transparently [36].

Central Processors: CPs

Central processors (CPs) are the fully capable (and most expensive) processors. They run all operating systems (including z/OS) and application software.

SAPs and ICF

System Assistance Processors (SAPs) functions as a part of I/O subsystem. These processors execute LIC (**Licensed Internal Code**), which technically incorrect is known as microcode or firmware. Similar **Integrated Coupling Facility** (ICF) processors also run LIC. ICFs provide memory and assist coordinating work when multiple mainframe systems co-operate in a so-called Parallel Sysplex. These "support processors" are undetectable for operating systems and applications.

IFLs, zAAPs and zIIPs

Integrated Facility for Linux (IFL), **z Application Assist Processors** (zAAPs), and **Integrated Information Processor** (zIIP) are CPs which have a few functions or instructions disabled by microcode. Hence they cannot run z/OS, but they are suitable for Linux, Java (also under z/OS), and eligible z/OS workloads (e.g. DB2) respectively. These processors are basically used to control and differentiate software costs. Substantial amount of money can be saved using these processors, and the IFLs partly explain the success of Linux on the mainframe. [57][35]

2.2.2.1 Capacity on demand, computation capabilities

The mainframe system allows for different forms of "**Capacity on Demand**". Additional processor can for instance be enabled on the run for a limited period of time to handle unexpected peak loads. On the other hand CPs can be "**kneecapped**" to operate at lower speeds in order to reduce software costs.

Notice that mainframe processors are *no* more capable than processors in more typical architectures. They stand out at some kinds of workloads but they fall behind at others. The strength is that they typically run at 90% utilization or more around the clock – mainframe systems can handle 100% utilization without problems (using prioritized queuing). Furthermore the work mainframe processors actually do is concentrated on core applications and OS execution. This is amongst other things achieved by letting help processors prepare and feed I/O and let specialized processor handle cryptography computation. [44]

With very large caches, huge memory sizes, hardware support for fast context switches and massive concurrent I/O capacity mainframes work well for typical "wide" business workloads (including transaction processing and large database management). Mainframes are in other words not for single task computation like weather modelling, protein folding, or rendering of 3D movies. These tasks are left to supercomputers and grid computing. [44]

2.2.2.2 Cryptographic Facility

The integrated cryptographic facility (CF) is an extension working as an integral part of a CPU. The CF "provides a number of instructions to protect data privacy, to support message authentication and personal identification, and to facilitate key management. The high-performance cipher capability of the facility is designed for financial-transaction and bulk-encryption environments, and it complies with the Data Encryption Standard (DES)." citation from [11].

2.2.3 Books, n-way systems

Newer mainframes are organized in "**multi-book**" setups supporting one to four books. Each book contains a MCM (**Multiple Chip Module**) with up to 12 or 16 of the above mentioned processors and up to 128GB memory. The books are interconnected with a super-fast bi-directional redundant ring structure, which allows the system to operate as a symmetrical, memory coherent, multiprocessor [36]. Then a z9 EC is equipped with multiple books, it is possible to remove and reinstall a single book during an upgrade or repair while the system is running. [57]

Mainframes are often referred to as **"n-way" systems**. The n refer to the number of processors (CP/IFL/ICF/zAAP) *not* counting the SAPs (the ones supporting the I/O subsystem). Currently the maximum is a **54-way** system, since such a system comes with 8 SAPs:

$$54 \text{ processors} + 8 \text{ SAPs} = 4 \text{ books} \cdot 16 \text{ processors} = 64$$

2.2.4 Storage (memory)

The terms "**central storage**", "**processor storage**", and formerly "**real storage**" are used for memory, the kind best comparable to RAM (Random Access Memory) on PCs. That is, the kind of memory a processor can access synchronously within an instruction.

"**Auxiliary storage**" or "**paging storage**" is physical storage external to the mainframe (disks, tapes). This type of storage is accessed asynchronously through an I/O request. This temporarily frees the processor to perform other task (rescheduling).

Data are moved between central and auxiliary storage by paging and swapping. This allows for virtual storage, where all users and separately running programs are assigned a unique address space, which in principle can as large as the architecture allows (64bits). Memory handling like this happens in the OS (operating system) layer.

2.2.5 Support element and HMC

Inside the mainframe box you find two IBM Thinkpad laptop computers. One constitutes the "**Support Element**" (SE) – the other one is failover component for the first. A Support Element provides communication, monitoring and diagnostic functions to the system.

Normally the SE is connected to a "**Hardware Management Console**", a desktop PC residing in more convenient surroundings than the data center. The HMC (or SE) can be used to configure hardware partitioning, and to monitor and control hardware like the processors.

2.2.6 Input/Output: Channels, channel subsystem

Figure 2-3 depictures a simplified system I/O configuration of a mainframe system. I/O-devices like disk drives, tape drives, and communication interfaces are connected through channels. **Channels** provide *independent* data and control paths between the I/O devices and storage (memory). This eliminates the need for the processors to communicate directly with I/O devices, which allows for concurrent data and I/O processing [11, p. 13-1].

In other words, the architecture implements DMA (Direct Memory Transfer) directly in the instruction set and *all* I/O devices are DMA devices equipped with advanced controllers to handle actual data transfer. This setup allows for thousands of I/O devices to be running at full speed without any noteworthy processor utilization. [75]

The communication links, which manages the flow of information to or from I/O devices, are also called "**channel paths**". Earlier "parallel-I/O interfaces" were used but these are now replaced by **"serial-I/O interfaces**" of **ESCON** or **FICON** types. [11, p-. 13-2]

The channel paths connect to **control units**, which contain the necessary logic for the channel subsystem to operate I/O devices in a uniform way. As illustrated switches between channels and control units can be used. These **switches** (or **directors**) allow sharing of control units and I/O devices across systems. Today control units, especially for disks, have multiple channel connections (possibly via a switch) and multiple connections to their devices. This allows for simultaneously data transfers on multiple channels.

The architectural addressing scheme limits the number of channels or "**Channel Path Identifiers**" (CHPIDs) to 256, which has proven inadequate on modern systems. To address the I/O requirements newer mainframes are equipped with **Logical Channel SubSystem** (LCSS), which gives four logical channels sets or a total of 1024 channels. A new **Physical Channel ID** (PCHID) layer has been invoked to represent the physical location of the I/O ports within the box. CHPIDs are then mapped to PCHIDs completely transparent to running programs (see “Channel I/O SubSystem” in Figure 2-3). [36]

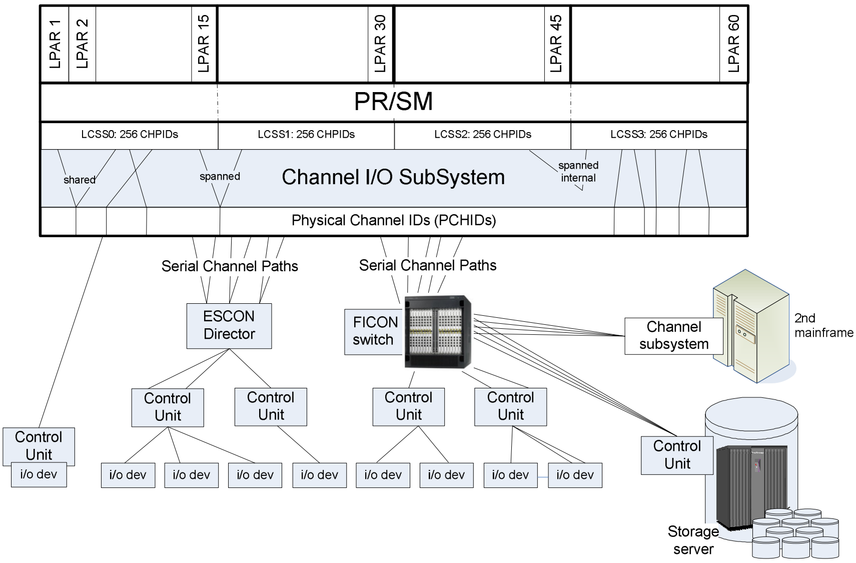


Figure 2-3: Simplicifed I/O configuration. I/O devices controlled by “Control units”, which are connected using FICON or ESCON channels (possibly through a switch or “director”). Physical Channel IDs are mapped transparently to the CHannel Path IDentifiers, which provides the data path from OS perspective.

Channels (and the connected control units and devices) can be ***shared*** between logical partitions / LPARs, which are self-contained and completely separated subsets of the machine running independent operating system (see section 2.2.7 below). Sharing of channels within an LCSS is enabled by the **Multiple Image Facility** (MIF). Channels can also be ***spanned*** across multiple LCSSs and thereby transparently be shared among the logical partitions within these LCSSs. This will however decrease the total number of channels. [36]

Within channels you, figuratively speaking, find **subchannels.** Subchannels are in-storage (in-memory) control blocks controlled/used by hardware representing I/O devices. A subchannel is provided for and dedicated to each I/O device accessible to the channel subsystem. They contain information about the I/O device connection, I/O operations and other associated functions concerning the device. This information is accessible for the CPUs by I/O instructions. [11]. Subchannels and used by the operating systems to pass I/O requests to the channel subsystem [36].

The architecture uses 16 bit sub channels addressing. Some sub channels are reserved for system use leaving 65,280 sub channels (per LCSS) for I/O devices on System z9 EC. [36]

2.2.7 PR/SM, LPARs and its configuration

Mainframes have a native hardware mechanism called **PR/SM** (**Processor Resource/System Manager**), which is used to divide the hardware into **Logical Partitions** (**LPARs**[1]). A LPAR is a *separate* subsets of the real hardware, capable of running an independent operating system. PR/SM is capable of sharing some hardware resources and distributing others between LPARs. PR/SM allows for up to 60 concurrent Logical Partition (as in Figure 2-3). The System z9 EC cannot even run in basic mode but solely runs in LPAR mode. PR/SM is designed to preserve (near) absolute reliability, which is considered far more important than fancy new functionally [6].

In some respects PR/SM can be compared with the hypervisor in virtual (VM) environments but it is, more accurately, a hardware mechanism that partitions the hardware rather than a software layer virtualizing it. Figure 2-4 illustrates how PR/SM share or divide the hardware (channels, storage, processors) into 4 Logical Partitions running z/OS or z/VM as operating systems.

As mentioned above, **channels** (and connected devices) can be shared between LPARs or they can be dedicated to LPARs. Some restrictions exist depending of the channel and device type. Up to 15 LPARs can be associated with each Logical Channel SubSystem (LCSS). There are in other words 256 channels (or CHPIDs) to be shared between or individually assigned to the LPARs in each LCSS group.

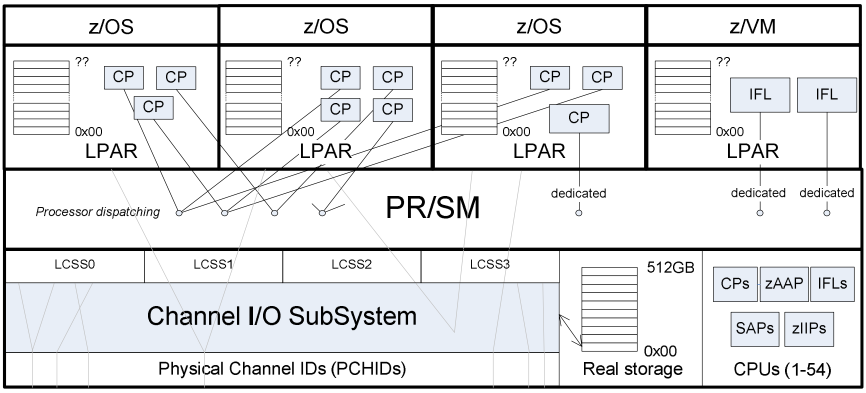


Figure 2-4: PR/SM (Processor Resource/System Manager) is the hardware mechanism, which natively split the hardware in self-contained sub-sets called LPARs (logical partitions). Storage (memory) is divided between LPARs, channels and processors can be shared or dedicated to LPARs.

Storage (memory) cannot be shared amongst LPARs. It is in other words impossible to create a common storage area accessible from multiple Logical

[1] According to [64] the correct abbreviation for an individual "Logical Partition" is "LP". "LPAR" is "Logical Partitioning", the "mode" or concept provided by PR/SM to create LPs. The more general approach of using LPAR for "logical partition" is adopted throughout this paper.

Partitions. Storage is divided into sections and remapped to location zero by PR/SM for every LPAR (as illustrated in Figure 2-4). Because the translation happens in hardware it is impossible for one LPAR to access or compromise the address space of another LPAR. [6].

PR/SM has an internal processor **dispatcher**, which can share real processors between multiple LPARs. A 1-way system can in principle run operating systems in several LPARs. It is also possible to dedicate processors to specific LPARs. A processor assigned to LPARs is called a logical processor. When using dedicated processors the physical processors are assigned to specific logical processor and thereby always available for the LPAR. Unused capacity in is on the other hand wasted. When using shared processors a logical processor can be dispatched to any physical processor.

Processing weights are used to divide shared CP capacity according to overall goals. LPARs with shared CPs can be “capped” to limit CP usage according to business goals or reduce impact on other LPARs.

The processor sharing of PR/SM is bound to result in some amount of overhead. When a physical processor is assigned from one LPAR to another it will always involve a context switch: The processor state (including register values, Program Status Word (PSW), accessible storage range, etc.) has be stored and restored as part of the process. The mainframe architecture includes several facilities to minimize this overhead.

The **SIE** (**Start Interpretive Execution**) instruction is probably the most important of these facilities. It enables fast and secure “context switches”, when a physical processor is dispatched to another logical processor. The SIE call provides a control block describing the “state” of the logical processor (register, etc.) and Dynamic Address Translation (DAT) structures needed for the switch. The use of SIE is briefly revisited in section 3.7.3 “Running z/VM in z/VM” on page 50.

All in all a Logical Partition makes up a completely separated mainframe environment capable of running any mainframe operating system, that is, if the LPAR configuration meets system requirements. The operating systems can hardly detect the difference of an LPAR and a “real system”. The z/OS operating system can however (if allowed) improve performance by dynamically shifting resources between LPARs. In this way z/OS extends PR/SM and enables sophisticated workload balancing.

PR/SM has an **Evaluation Assurance Level 5 certification** - the highest grade yet given following a “Common Criteria” security evaluation (in accordance to international standards). EAL5 is probably the best proof given, that workloads are completely separated and unintended flow of information between logical partitions is impossible [19]. As of September 2006 the IBM System z, was the only systems with the prestigious assurance level for partitioning [1], making is acceptable for running concurrent governmental and/or military workloads, which normally requires physically separated servers with no connections to other systems.

2.2.7.1 System configuration

The I/O configuration is specified in **IOCDS** (the I/**O Configuration Data Set**). IOCDS is created by **IOCP** (**I/O Configuration Program**), which runs under z/OS, z/VM, or stand alone on a empty system). IOCDS is placed on the Support Element hard disks and used on **POR** (**Power On Reset**), where information about the configuration is placed in the **Hardware System Area** (**HSA**) to initialize the hardware. HSA is the lower part of main storage (memory), which contains tables reflecting the current system. Figure 2-5 tries to picture this.

Today most resources can be (re)allocated/(re)configured dynamically without power-on-reset (POR) or booting / **initial-program-load'ing (IPL)** the operating systems. The **HCD** (**Hardware Configuration Definition**) program greatly simplifies this process. With HCD it is possible to have a single **IODF** (**I/O definition file**), which contain all relevant I/O configuration including data about the Channel SubSystem, logical partitions, processors, and even external switches, control units and devices. Actually the same IODF can contain information about several mainframe boxes.

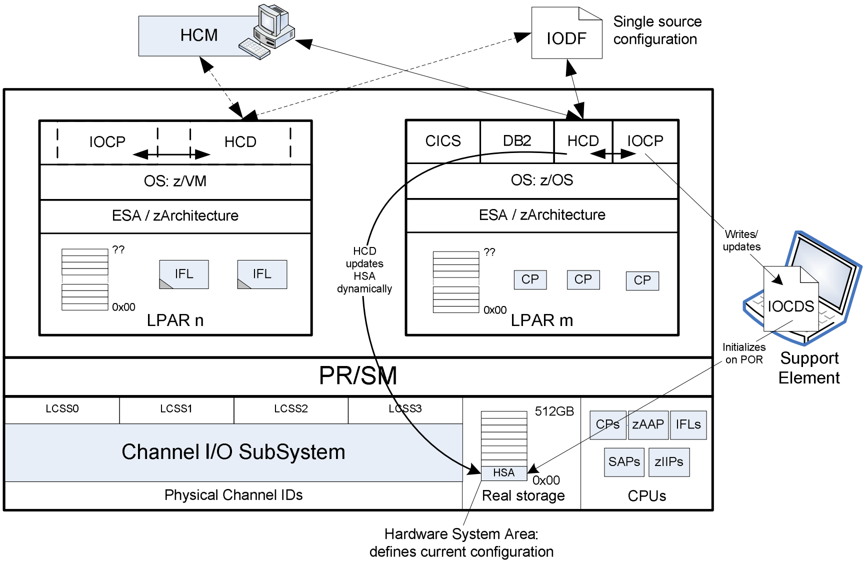


Figure 2-5: Hardware configuration is loaded into the "Hardware System Area" from the Support Element on Power-On. HCD (possibly the HCM GUI) can be used to make changes dynamically.

Very conveniently HCD is capable of validating entered configuration data for consistency and completeness to avoid many errors. But even more important, it is capable of dynamically changing the current configuration and simultaneously updating the IODF. The latter makes is possible to invoke

IOCP, write IOCDS based on the IODF and with that initialize the system on next Power On Reset. In other words this ensures that dynamic changes are reactivated after a POR.

The IODF is an important combined configuration source, since it also supplies the configuration data, which the z/OS operating systems needs for device configuration during Initial Program Load (IPL).

Finally HCD functions as a server interface for the PC based client program **HCM** (**Hardware Configuration Manager**). HCM has a graphical user interface, where HCD simply supplies an interactive text based interface. All configuration changes made in the GUI are still fully validated by HCD to avoid system outages due to errors. But in addition to the logical aspects of hardware configuration HCM also manages physical infrastructure aspects like cabinet and cabling.

[15] [13] [25] [4]

2.2.8 Hard drives: Direct Access Storage Devices

In mainframe terminology a hard drive is a **DASD** (**Direct Assess Storage Device**). Besides DASD the terms "**disk volume**", simply "volume", "disk pack", and "Head Disk Assembly" (HAD) are some times used when referring to disk drives.

The conventional DASD storage architecture is **ECKD** (**Extended Count-Key-Data**). ECKD is a refinement of CKD (Count Key Data) optimized for non-synchronous DASD control units. Such disk devices are divided into cylinders, which contains tracks. (Historically a single magnetic read/write head could access one track on a disk plate per revolution. As disk plates where "stacked" it was possible to read/write several tracks/plates at once. The tracks concurrently accessible without repositioning the access mechanism make up a cylinder). The tracks contain variable length record each containing a count field (cylinder number, head number, record number, length of data); typically a key field (search argument); which is following by the actual data.

The ECKD DASD category covers a variety of different physical medias. The IBM **3390** disk and the **3990 control unit** are the most used disk devices today. The first models were released in the late 1980s. Newer models with higher capacities with named "3390 model (3, 9, 27)" have been introduced later.

Today real physical 3390 disk and 3990 control units are outdated and replaced by large storage servers actually emulating the old technology to maintain backward compatibility. DASD devices (emulated or not) are connected to the mainframe using the typical I/O setup: though a control unit attached to one or more I/O channels using ESCON or FICON connections. The control unit can be an integrated part of the storage server. Such a setup is illustrated bottom right in Figure 2-3 on page 11. [23]

Enterprise Storage servers typically use multiple high-end RISC processors to handle control unit and device emulation. They are equipped with considerable **cache memories** (in the order of 8 to 32GB). Naturally new hardware functions have been introduces over the years to boost performance. On the software side these functions are introduced as OS extensions, which ensure that new technologies are exploited by old applications and backwards compatibility is maintained.

In the world of z/VM virtualization and Linux, industry standard **SCSI** devices in a **Storage Area Networks** (**SAN**) have also been introduced. The disks are connected via SCSI over **FCP** (Small Computer System Interface over **Fibre Channel Protocol**).

These DASD utilize another data storage architecture called **FBA** (**Fixed Block Architecture**), which is quite different from ECKD. FBA stores data in fix-length blocks (512 byte). Blocks are addressed/accessed using block numbers, which simply are assigned consecutively from the beginning to the end of the disk.

[59] [57] [66][8]

2.2.9 Network connectivity & Hypersockets

Mainframes connect to industry standard LAN networks using **OSA** (**Open System Adapter**) technology. System z9 EC features OSA-Express and OSA-Express-2 supporting e.g. 10 Gigabit Ethernet, 10 Gigabit Ethernet and 1000BASE-T Ethernet (10/100/1000 Mbps). [79]

Open System Adapters (OSA) is basically an advanced Network Interface Card, connected via channels like other I/O devices. A separate channel and CHPID exist for every connection to the “open world” (for example for each Gigabit Ethernet port). An Open System Adapter may be shared between several LPARs (and VM guest). [59]

(Most) OSAs can run in two modes: QDIO and non-QDIO. “Queued Direct Input/Output” is very efficient mechanism to transfer data. Special memory queues and a signalling protocol are used to directly exchange data between the OSA-Express microprocessor and TCP/IP stacks in the operating systems. It basically bypasses the normal Channel I/O subsystem and thereby reduces system overhead and SAP (System Assist Processor) utilization in particular. Non-QDIO basically uses the normal I/O path (control unit, channels, SAPs). [59]

The OSA technology generally provides many functions to offload the system (TCP/IP stack in OS; processors and I/O SubSystem). In Layer 3 mode the QDIO microcode can offload the TCP/IP stack for IP processing of: multicast support, broadband filtering, building MAC and LLC headers; and ARP processing. Similar can TCP/UDP and IP checksum calculation be handled by OSA Express hardware on behalf of the TCP/IP stack in Linux and z/OS.

Hypersocket technology provides fast TCP/IP communications between the logical partitions and virtualized environments on a mainframe box connecting them by internal "**virtual LANs**". The communication runs though the system memory minimizing latencies and maximizing bandwidth. Each HyperSockets LAN occupies a Channel Path ID (CHPID). [79]

HiperSockets are implemented in microcode (Licensed Internal Code, LIC) and emulates the Logical Link Control (LLC) layer of an OSA-Express QDIO interface. Because of this relationship, HiperSockets are sometimes referred to as **internal QDIO** (**iQDIO**). [78]

2.2.10 Terminals, TN3270

"**Dumb terminals**" or "**green screens**" have characterized mainframe usage from the beginning of the mainframe era. The original terminals had just enough computing power to fetch and display a full screen of text, and more important receive "whole screens" of user input, allowing the user to input multiple values before communicating with the server, thus saving processor cycles.

These terminal principles are still used heavily today - "**3270**" is still the primary user interface. For many years the dedicated hardware terminals have been replaced by terminal emulators for instance running under Windows. The protocol used today is **TN3270**, which is a blend of telnet and the 3270 terminal protocol. TN3270 took over as TCP/IPs became popular.

2.2.11 Other technologies

Mainframe systems include many other technologies, which distance them from other servers. This includes different "clustering technologies", which enables high availability.

CTC rings are relatively simple clustering technique connecting multiple systems or logical partitions using channel-to-channel (CTC) communication (connection between two CHPIDs). The CTC-ring can be used to exchange control information like usage and locking info on shared disk systems. Similarly the CTC ring can be used to share job queues and security information.

Parallel Sysplex is a more capable clustering technique. It depends on the mentioned ICF processors (Integrated Coupling Facility) to create Coupling Facilities. Independent mainframe boxes or logical partitions (LPARs) can be used for CFs. They normally have a large memory and mainly provide locking information to the attached systems, cache information and data lists. Multiple boxes in a sysplex appear as a single large system.

Geographically Dispersed Parallel Sysplex (GDPS): Is a kind of extension of the parallel sysplex for multi site enterprises. It primary provides disaster recovery and continuous availability solution. Critical data is mirrored and workloads are efficiently balanced between sites. GDPS uses automation and Parallel Sysplex technology to help manage multi-site databases, processors, network resources and storage subsystem mirroring.

2.3 Operating systems

A handful of operating systems dominate in the mainframe world, each system with different purposes and characteristics. The two operating systems of particular interest in this context, z/VM and Linux for zSeries, are described in chapter 3 and 4 respectively. The following shortly introduces the other main players.

z/OS

z/OS is the most widely used mainframe operating system of all. It is designed with special consideration to stability, security and continuous availability to an extent, which makes other operating systems pale by comparison. It has its origin in OS/360, which over time has evolved and been know as OS/390 and MVS (Multiple Virtual Storage, in several versions).

Today z/OS supports UNIX APIs and applications; it runs Java and communicates via TCP/IP and the web.

z/VSE

“Virtual Storage Extended” is typically used on smaller mainframes. The z/VSE OS provides a smaller and simpler base for batch and transaction processing than z/OS. According to IBM “z/VSE is excellent for running routine production workloads consisting of multiple batch jobs (running in parallel) and extensive, traditional transaction processing”.

z/VSE has its origin in DOS/360 (Disk Operating System), the first disk-based operating system for System/360. Its simplicity and small size has kept it alive, although it originally was a temporary measure until OS/360 was finished. It developed into DOS/VS (virtual storage), VSE/SP, VSE/ESA, and then finally z/VSE.

z/TPF

z/TPF or “z/Transaction Processing Facility” is a special-purpose system for very high transaction volumes. It is used by credit card companies and airline reservation systems. It was formerly known as Airline Control Program (ACP). It loosely couples multiple mainframes to handle thousands of transactions per second running uninterrupted for years.

[57]

3. IBM z/VM

3.1 Presentation

z/VM is an mainframe *operating system*, which has the capability to create hundreds or potentially thousands of virtual mainframes (*virtual machines*, VMs) inside a single mainframe (actually inside a logical partition). Each virtual machine supports the full architectural instruction set and is in principle capable of running any mainframe operating system. Actually z/VM is even able to simulate instructions and thereby offer a system architecture, which in reality is unsupported by the actual hardware.

The basic concept behind z/VM is different from the LPAR concepts of PR/SM although many similarities do exist. The flexibility of z/VM is much higher: New virtual machines can e.g. be defined and IPL'ed (Initial Program Loaded, "booted") without problem on a running system. There is in principle no limit on the number of virtual machines whereas the current maximum of LPARs is 60. The abilities and performance of the individual virtual machines naturally depend on the actual hardware and resources, which z/VM is assigned to share transparently between them.

z/VM has many purposes: It can be used to test new operating system releases in a controlled environment. It can be used to install and test program fixes concurrent with production running. It can in principle contain a complete disaster recovery environment for big complex mainframe setup including simulation of coupling facilities in a parallel sysplex configuration. It provides a simple single user operating system (called CMS), which can handle thousands of current users, e.g. allowing them to develop and run their own programs.

Lately z/VM has endured a renaissance providing the perfect platform Linux on zSeries. This enables large server consolidating project and big saving in TSO (Total Cost of Ownership) according to the advocates of the environment.

3.2 History

The first product in the z/VM product line (Figure 3-1) was "Virtual Machine Facility/370" or "VM/370", which was shipped in 1972. The committed users and supporters of z/VM have recently (August 2007) celebrated the operating systems 35th anniversary. "VM" has a long and fascinating history specially caused by its somewhat troublesome childhood, where it lived on the very edge of the computer technology development.

VM/370 was based on CP-67 and CMS, which again was based on CP-40: a system conceived in 1964. CP-40 was strongly influenced by CTSS, the Compatible Time-Sharing System. CTSS pioneered general purpose time-sharing systems in the early 1960s and also constituted basis for MULTICS. [55]

The birth and development of “VM” was highly influenced by IBM’s decision not to include hardware for dynamic address translation in System/360 to enable virtual memory capabilities. This led the MIT time-sharing project “MAC” to turn to GE (General Electrics) for development of MULTICS. Similar Bell Labs found another vendor to begin the system, which was to become UNIX.

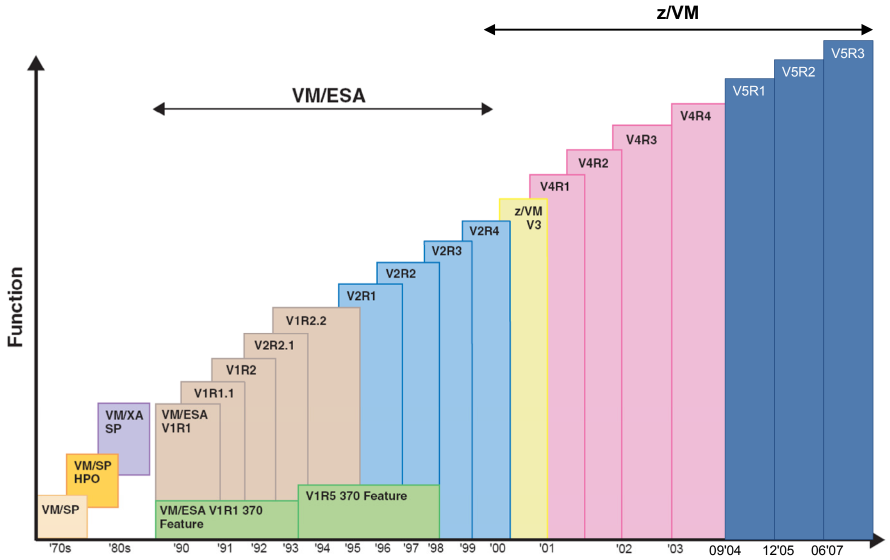


Figure 3-1: The evolution of VM (graphic from [16] modified according to [46]).

As a result the people at “Cambridge Scientific Center” (CSC), which were supposed to be the centre of IBM’s time-sharing activities at MIT, had nothing to do. The people at CSC decided to create their own time-sharing system for System/360, which turned into CP-40 and CMS “the conceptual elements of z/VM.

IBM discovered the significance of loosing the “MAC” project in regards to time-sharing-system. They decided to equip the following S/360-67 with address translation and developed software called “TSS” (Time Sharing System). TSS was later abandoned before initial stability and performance problems were solved.

The CP/CMS project miraculously survived in spite of IBM unwillingness. This was a result of a complex interplay of factors including ingenious funding strategies and the system having functionally of no other system.

The history of z/VM could constitute a study of its own. In this context it should basically be noticed that z/VM is a highly mature product and its performance and stability is a result of 35 years of continuous use and development. The CP-40 work and principles were ground-breaking and defined the concepts behind newer virtualization environments like VMWare and Microsoft Virtual Server.

[55][74]

3.3 Virtualization concepts and principles

Virtualization is a broad term covering many different concepts and techniques. Virtualization is, in general terms, the ability to abstract computer resources into logical or virtual resources and/or computing environments.

Virtualization covers any technique, which can be archieved using resource sharing, resource aggregation, emulation, and insulation. **Resource sharing** is the ability to create multiple virtual resources based on a physical resource e.g. by partitioning or time sharing. **Resource aggregation** is ability to combine multiple physical resources into fewer virtual resources. With **emulation** available physical resources are used to imitate or simulate resource types and features, which are physically unavailable. The last cornerstone of virtualization is **insulation**: the ability to segregate resources and/or environments and render it impossible for them to influence each other.

Together these four fundamental capabilities enable many kinds of virtualization. The area of **resource virtualization** covers for example several techniques, which are not immediately associated with virtualization:

- Combining disk into large logical disks (RAID and Volume managers)
- Combining multiple discrete computer into a whole (grids, clusters)
- Creating virtual networks within another network (VLAN, VPN).

[66] [45]

3.3.1 System virtualization: creating virtual machines

In this context the focus is on “system virtualization” or “platform virtualization” – the ability to create multiple “*virtual machines*”, which in every detail architecturally resemble a hardware platform - in this case the mainframe. It is the ability to transparently share resources without the consumers’ knowledge to archive better resource utilization.

Two main approaches for system virtualization exist: “Hardware partitioning” and “hypervisor based” technology.

3.3.1.1 Hardware partitioning

Some computer platforms support “hardware partitioning”. They allow for the hardware to be partitioned using coarse grained units (whole system boards, processors, etc.). Each partition runs a separate operating system. This approach allows for hardware consolidation but lacks the ability to share resources in order to increase utilization. “Sun Domains” and “HP nPartitions” are examples of this technique. [21][66][34]

3.3.1.2 Hypervisor based partitioning

Virtualization by “hypervisors”, on the other hand, allow for fine-grained, and dynamic sharing of resources. A hypervisor constitutes a shallow software layer (possibly within firmware) used to create and control virtual machines. Hypervisors are also called “*Virtual Machine Monitors*” (VMM) [69][77]. Generally two main types of hypervisors exist.

Type 1 hypervisors run directly on system hardware or in system hardware as firmware / millicode / microcode. When running on hardware they act as a low level operating system creating “virtual machines” for other operating systems. The principle is illustrated in Figure 3-2, page 22. Type 1 examples include WMware ESX Server, Xen, z/VM, and the earlier mentioned “PR/SM” (hardware implemented). [21]

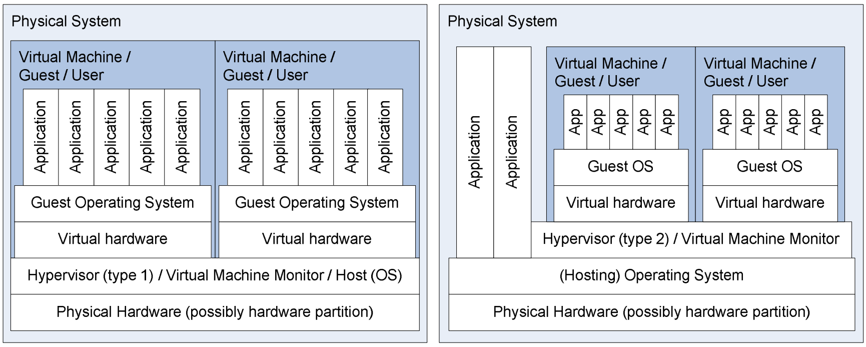

Physical System
Virtual Machine / Guest / User
Application
Guest Operating System
Virtual hardware
Hypervisor (type 1) / Virtual Machine Monitor / Host (OS)
Physical Hardware (possibly hardware partition)
Physical System
Application
Virtual Machine / Guest / User
App
Guest OS
Virtual hardware
Hypervisor (type 2) / Virtual Machine Monitor
(Hosting) Operating System
Physical Hardware (possibly hardware partition)

Figure 3-2: Hypervisor based system virtualization: Type 1 hypervisor (left) and Type 2 hypervisor (right). Notice basic virtualization terminology.

Type 2 hypervisors run inside an operating system as any other application program. The virtual machines running **guest operating systems** live inside the application (see Figure 3-2). Type 2 hypervisors are also known as “hosted” hypervisors [66]. The operating system *hosting* the hypervisor provides the basic services like I/O device support and memory management. Type 2 examples include “Microsoft Virtual Server”, “VMware GSX Server”, and “Win4Lin”. [21] [66]

3.3.2 Variant types of hypervisor based virtualization

Distinguishing Type 1 and Type 2 hypervisors does however not suffice, when characterising hypervisor technologies. Multiple techniques are used within each of these main categories. The differences arise from varying levels of virtualization support within hardware and from varying levels of interaction between hypervisor and guest operating system. The following sections present important hypervisor techniques, but first a few low level computation fundamentals are recapped.

Multitasking operating systems run multiple processes and shield them from each other. To ensure the operative system remains in control CPUs typically supports two modes of operation: **unprivileged** (or **user**) mode and **privileged** (**kernel** or **supervisor**) mode. Only the operating system running in privileged mode is allowed to run **privileged instructions**, which are instructions that can change the overall state of the system.

When an unprivileged (no OS) process executes a privileged instruction it causes a **trap** (an exception), which forces the CPU back into the operating system code. The operating system handles the situation (in privileged CPU mode) and returns control to application process afterwards. There are other situations, which drive a CPU from user mode to kernel mode: for instance (timer) **interrupts** and **page faults**.

Under normal conditions privileged instructions are avoided in applications programs. Instead they depend on **syscalls** (operating system functions), which deliberately switches into **kernel mode** and handles the operating in a more efficient manner avoiding traps.

[71]

Trap and Emulate

"Trap and emulate" is the basic virtualization method, which was used by mainframes in the 1960s and 1970s for instance by z/VM predecessor VM/370. The hypervisor runs in privileged mode while the guest operating systems within the virtual machines run in user mode. When a guest operating system issues a privileged operation (e.g. an I/O operation) it is trapped and handled by the hypervisor. The hypervisor emulates or simulates the operation, which opens for resource sharing and for taking other virtual machines into consideration.

This method does allow for running completely unmodified guest operating system. Unfortunately it also introduces a substantial overhead attributed to both traps and emulation. Examples of use: CP-67 and VM/370.

This method cannot be used on all architectures. It is vital that all privileged operations trap into kernel mode in order for the hypervisor to emulate a trustworthy virtual machine. Traditional x86 processors have several instructions, which simply behave differently according to the mode of operation. For example an instruction exist, which simply tell the actual mode of operating. Since no trap occur it is impossible for the hypervisor to fake an answer of privileged mode, which the guest operating system would expect.

[71] [66]

Translate, trap, and emulate

The second method is almost identical to the "trap and emulate" method above. This method incorporates an extra **translation** step, which replaces privileged operations within the guest's binary OS kernel code with special “**explicit trap operations**”. This technique is also knows as **binary translation**.

User mode applications program within the guest operating system still run natively in unprivileged CPU mode. The kernel code of the guest operating system is however split into **basic blocks** (blocks of consecutive instructions separated by flow control instruction like branch, jump or return). Before execution these basic blocks are examined for privileged operations and if such are found they are replaced with specific hypervisor calls.

The hypervisors using this technique reduce the overhead combining “safe” basic blocks into larger blocks and then reuse already “translated” code. In some situations this approach is actually faster than “trapping”. As an example the earlier mentioned instruction, which is used to determine the current CPU operation mode, can simply be substituted by an instruction, which loads a constant corresponding to “privileged mode”. Over time this is certainly faster than “trapping” and emulating the instruction within the hypervisor.

This method resolves in other words the “missing trap problems” of the x86 platform. “Translate, trap, and emulate” is therefore the technique behind VMware and Microsoft Virtual Server, which runs on the x86 platform.

[71] [66]

3.3.2.1 Paravirtualization (Hypervisor call method)

Sometimes a hypervisor creates virtual machines with an architecture, which differs slightly from the physical machine architecture. As a consequence parts of the operating system running in such a virtual machine have to be rewritten to run in the special virtual environment. The changes typically include explicit **hypervisor calls** for I/O and memory management. This kind of virtualization is known as **paravirtualization.**

In other words, special code segments, which explicitly call the hypervisor functions, are introduced to the source code of the guest operating system. This allows for better optimization of the interaction between hypervisor and guest. The operating system becomes aware of the virtual environment. Therefore this virtualization strategy improves scalability; it reduces system complexity, and allows for high efficiency. Unfortunately this method cannot be applied to proprietary operating systems, if the vendor chooses not to support the virtualization platform.

The method was already applied in VM/370 and it is still used in z/VM today. A newer example is Xen, which among other operating system runs Linux, OpenBSD, FreeBSD, and OpenSolaris.

[66][77]

3.3.2.2 Direct Hardware Support

The last hypervisor variant to be mentioned here is relies on direct virtualization support in hardware. In this case the architecture includes special instruction and hardware constructs to run virtual machines very efficiently.

This can for example include a special "guest" CPU mode. In this mode the guest operating system can run most instructions (including many privileged). The hardware assists when processors are dispatches to virtual processors and when execution is handed back to the hypervisor. This can also provide the hypervisor with better measures to accommodate the conditions, which causes exceptions and hand back control to the hypervisor.

Other examples of virtualization mechanism in hardware include the ability to route I/O interrupts directly to the guests; optimizations for multiple levels of virtual memory (multiple levels of dynamic address translation); and hardware measures to keep the hypervisor immune for critical errors (including I/O related errors) within the guest.

The mainframe is the classical example having provided hardware "assist" since the "370 days". z/VM (and PR/SM) utilizes all these capabilities. The last few years of increased focus on virtualization have also made Intel and AMD integrate (incompatible) virtualization technology in newer x86 chipsets and processors.

[65][66][48]

3.4 Main z/VM components and functionality

To summarize: z/VM is an *operating system*, which enables *system or platform virtualization*. It shares physical mainframe resources and abstracts them into *virtual machines*, each capable of running an independent operating system. z/VM runs directly on the hardware and is therefore a *type 1* hypervisor. It exploits both *paravirtualization* and *direct hardware virtualization support* to enable efficient virtualization; later topics will expand on this. The following sections introduce the main components and features of z/VM.

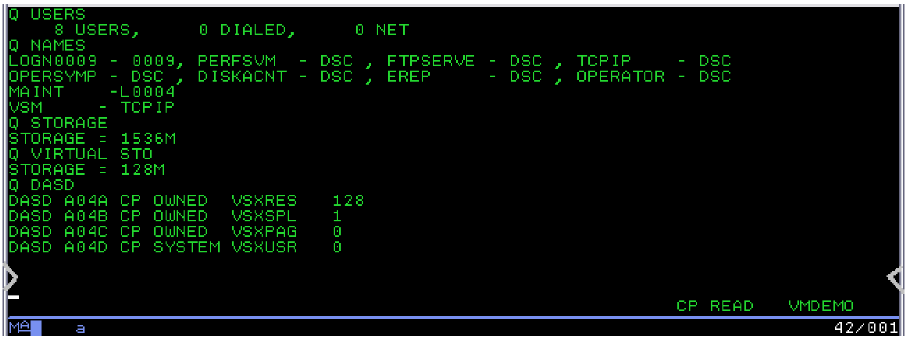
3.4.1 CP - the control program

The absolute cornerstone of the z/VM OS is called "CP" – the Control Program. CP is the *hypervisor* or *VMM* (Virtual Machine Monitor) of z/VM. CP is responsible for creating and managing virtual machines, for allocating resources to virtual machines or sharing resources between them.

CP contains different sub systems or components, which allow for it to function as a hypervisor. It contains for example a scheduler and a dispatcher component, which enable processor sharing. The paging sub system allows for over commitment of main storage by migrating memory pages to and from extended storage and disk. CP also contains measures, which can enable communication between virtual machines. Another vital part of CP is the login screen, which typically is the first screen you meet, before being able to utilize the system directly (opposite utilizing the system "indirectly" e.g. by accessing a virtual machine running Linux via SSH). Most of these topics will be revisited later.

Although CP constitutes the operating system part of z/VM, it should be noticed that CP first of all is a "virtual machine handler". It is *not* a full fledged operating system like Linux, Windows Vista, or even MSDOS. It has no file system functionality (for ordinary use) and no convenient methods for loading and running programs. z/VM does supply such facilities though CMS, but CP is simply a resource manager. [66]

CP provides a 3270 console interface to manage virtual machines and resources. This interface can for example be used to dynamically attach or detach resources to running virtual machines. It can also be used to enquire CP for information about the system and virtual machines. Queries are issued within virtual machines. CP commands can be entered directly, when the virtual machine runs in **CP mode** - without an operating system (Figure 3-3). CP commands can also be issued directly from CMS (the single user operating system within z/VM) and actually from within any guest operating system by prefixing the command with `#CP`.



```
Q USERS
    8 USERS,     0 DIALED,     0 NET
Q NAMES
LOGN0009 - 0009, PERFSVM  - DSC , FTPSERVE - DSC , TCPIP    - DSC
OPERSYMP - DSC , DISKACNT - DSC , EREP     - DSC , OPERATOR - DSC
MAINT    -L0004
VSM     - TCPIP
Q STORAGE
STORAGE = 1536M
Q VIRTUAL STO
STORAGE = 128M
Q DASD
DASD A04A CP OWNED  VSXRES   128
DASD A04B CP OWNED  VSXSPL   1
DASD A04C CP OWNED  VSXPAG   0
DASD A04D CP SYSTEM VSXUSR   0

                                                            CP READ   VMDEMO
MA     a                                                               42/001
```

Figure 3-3: Virtual machine running in CP-mode (without operating system). CP responds to basic enquiries about the actual virtual machine and the running system.

3.4.2 CMS - Conversational Monitor System

The second primary component of z/VM is the "**Conversational Monitor System**" or simply CMS. It is a single user operating system, which functions as a shell/console interface to z/VM. CMS is the *default* operating system developed and optimized to live inside the virtual machines provided by CP. It depends on paravirtualization techniques (special hypervisor calls) to an extent where is has become incapable of running natively – without CP (see section 3.7.1 "Hypervisor calls, Diagnose instructions" on page 47).

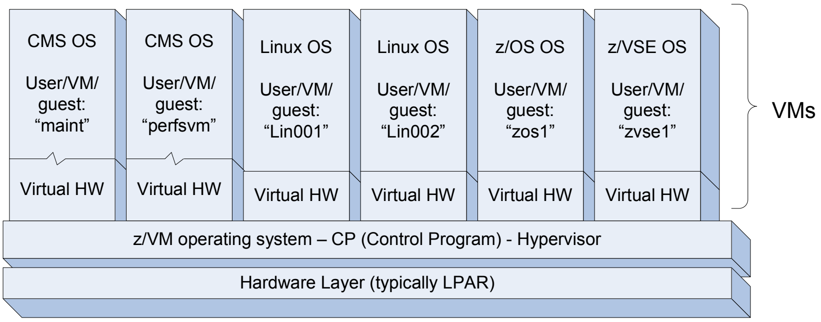
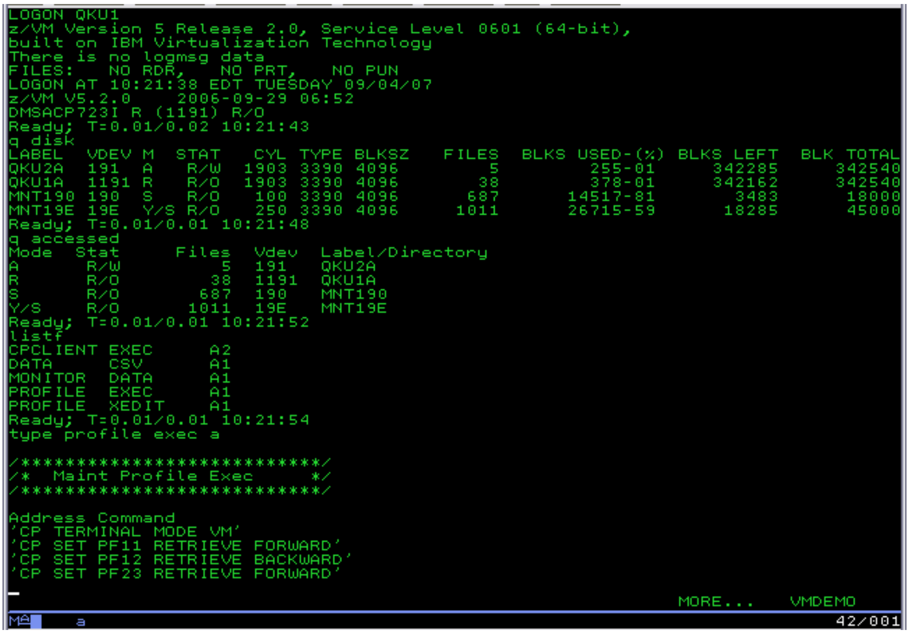


Figure 3-4: CMS is a small operating system intended to serve a single person per instance. It is provided as an integral part of z/VM and runs in a virtual machine like other guest operating systems like Linux, z/OS and z/VSE. The virtual hardware is depictured a little different to illustrate that CMS are dependent of special instructions (hypervisor calls) provided by CP.

CMS is intended to facilitate the virtual machine administration and configuration tasks within z/VM. But it is much more than an administration and configuration tool. It is designed to provide a large number of users with an interactive interface and the ability to do a variety of different task. CMS supplies an **Application Programming Interface** (API) and with that basis for program development and execution. Many programming language environments are supported (not all by default): Ada, Assembler, C, C++, COBOL, FORTRAN, Pascal, PL/I, and REXX. [7]

The native user interface of CMS is naturally a 3270 console. The console interface itself appears in several disguises. The most basic interface constitutes a simple command line interface (Figure 3-5). Some CMS programs and tools provide more interactive or “menu driven” interfaces. CMS shell itself can also run in **Full Screen Mode**, which enables scroll back functionality, a status area, and more intuitive use of functions keys.

CMS includes many “programs” / “tools” / “commands” for solving routine tasks. These include the `XEDIT` file editor to create, modify, or manipulate CMS files; `FILELIST` to mange files; and `HELP` the z/VM Help Facility, which provides command syntaxes, task oriented guides and a glossary among many other functions.



```
LOGON QKU1
z/VM Version 5 Release 2.0, Service Level 0601 (64-bit),
built on IBM Virtualization Technology
There is no logmsg data
FILES:    NO RDR,    NO PRT,    NO PUN
LOGON AT 10:21:38 EDT TUESDAY 09/04/07
z/VM V5.2.0    2006-09-29 06:52
DMSACP723I R (1191) R/O
Ready; T=0.01/0.02 10:21:43
q disk
LABEL  VDEV M  STAT   CYL TYPE BLKSZ   FILES  BLKS USED-(%) BLKS LEFT  BLK TOTAL
QKU2A  191  A   R/W  1903 3390 4096        5        255-01     342285     342540
QKU1A  1191 R   R/O  1903 3390 4096       38        378-01     342162     342540
MNT190 190  S   R/O   100 3390 4096      687      14517-81       3483      18000
MNT19E 19E  Y/S R/O   250 3390 4096     1011      26715-59      18285      45000
Ready; T=0.01/0.01 10:21:48
q accessed
Mode  Stat      Files  Vdev  Label/Directory
A      R/W          5  191   QKU2A
R      R/O         38  1191  QKU1A
S      R/O        687  190   MNT190
Y/S    R/O       1011  19E   MNT19E
Ready; T=0.01/0.01 10:21:52
listf
CPCLIENT EXEC      A2
DATA     CSV       A1
MONITOR  DATA      A1
PROFILE  EXEC      A1
PROFILE  XEDIT     A1
Ready; T=0.01/0.01 10:21:54
type profile exec a

/****************************/
/*  Maint Profile Exec      */
/****************************/

Address Command
'CP TERMINAL MODE VM'
'CP SET PF11 RETRIEVE FORWARD'
'CP SET PF12 RETRIEVE BACKWARD'
'CP SET PF23 RETRIEVE FORWARD'
_
                                                                 MORE...     VMDEMO
MA      a                                                                    42/001
```

Figure 3-5: Virtual machine running CMS – the single user operating system within z/VM. CMS responds to enquiries about available disks, accessible disks, the content of default disk A and content of a specific file.

CMS is also capable of running job in batch mode. It can be used to share data between CMS users and other systems. Not to mention the facilities to communicate with other system users.

[66]

3.4.2.1 CMS file system

Being an actual operating system in contrary to CP, CMS has file system support. All CMS files are record-oriented: The file management routines always write files in fixed physical blocks. This is done regardless whether it is fixed- or variable-length records. Fortunately is rarely necessary to specify either logical record length+record format or block size when creating a CMS file.

CMS files are stored on minidisk (virtual device representing a section of a real DASD) or within Shared File System (SFS). A **minidisk** is a flat structure, which holds a number of files. When using the **Shared File System** files are stored in a file pool, which typically is a large amount of DASD space, which contains files for many users. A user needs to be *enrolled* in a file pool to be able to use it. In contrast to minidisks, SFS supports a hierarchical structure, where files are stored in directories or sub directories like most other file systems.

Files are named using a **file identifier** (**file ID**), which consist of three parts: 1) File Name “fn”**,** 2) File Type “ft”, and 3) File Mode “fm” or Directory

Name “dirname”. Both File Name and File Type can be one to eight characters long, and basically correspond to filename and extension in more other operating systems. The third part, “File Mode”, actually represents the place (the disk) where the file is stored, since mini disks are made available to CMS by assigning them a “File Mode” character. In some degree File Mode resembles a dynamic version of the drive letters in Microsoft Windows and MSDOS.

[7] [66]

3.4.2.2 Byte File System, BFS

CMS has a POSIX support option called “z/VM OpenExtensions”, which includes another file system “**Byte File System**” (BFS) using another type of files: BFS files. Similar to the UNIX operating system, BFS files are also organized in a hierarchical structure made up by directories. BFS files are byte-oriented, rather than record-oriented. It is possible to copy BFS to CMS record files and vice versa. [66]

3.4.2.3 CMS as Boot Loader

Another frequent use of CMS is as an advanced boot loader. It helps loading the actual operating system; as LILO or GRUB often is used to load Linux on personal computers. E.g. when Linux runs under z/VM, CMS is often used to prepare a virtual disk (“RAMDISK”) to be used as Linux swap disk, before the Linux OS is started. When the actual operating system is loaded it replaces CMS, since two operating systems cannot coexist side by side. [66]

3.4.3 Users and their privileges

As mentioned earlier z/VM was originally developed to provide a high number of users (as in persons and individuals) with a virtual environment, which was architectural equivalent to a real computer system. A virtual machine is therefore a **user** in z/VM terminology. *The terms “**user**”, “**guest**” and “**virtual machine**” is therefore used interchangeably.*

The consequence is that there is no conceptual difference between a system programmer’s or system administrator’s user (running CMS) and a virtual machine running z/OS or Linux. They are all virtual machines, all z/VM users, they all defined in the *user direct*ory (see section 3.5.2), and they all “receive” a virtual environment on login.

The available resources inside the virtual machine might vary a lot: The system programmer only needs a smaller amount of storage (memory) to run the CMS operating system, but requires access to CMS disks with system tools and configuration files. A Linux guest, on the other hand, requires more storage, a good chunk of disk space, and often access to network devices, but it has no particular reason to access CMS disks.

3.4.3.1 Privilege classes

z/VM uses **privilege classes** to control the individual users capabilities within the system. A single letter or number is used denote a privilege class.

The letters A to G are defined by default. Class G is used for general users without any special privileges beyond functionally to control their own VM. Classes A to F defines certain administrator roles. The class B is for instance the “System Resource Operator”, which controls all the real resources of the z/VM system (except those controlled by the system operator and the spooling operator). Users are assigned at least one privilege class but can also be assigned several.

[27]

3.4.4 Architectural support

The virtual machines created by CP mimic one of several different architectures. The operating modes are denoted ESA, XA, and XC.

“ESA” virtual machines behave exactly like the 31 bit ESA/390 architecture. These machines can furthermore be switched into the newer 64-bit *z/Architecture*. This is done by the guest operating system, which issues a specific instruction doing so. Older systems exploiting ESA/370 architecture and even older 370-XA systems can in most cases run in ESA/390 mode because of backwards capability.

“XA” virtual machines are actually functionally equivalent to the “ESA” type above. It is supported for compatibility since some CMS applications require to be running in XA mode.

“*XC*” virtual machines behave according to the “*Enterprise Systems Architecture/Extended Configuration”* (ESA/XC). This specific architecture is exclusively available in z/VM virtual machines as provided by CP. ESA/XC differs from the other architectures by providing special services only relevant for applications inside virtual machines. It allows for virtual machines to create and share multiple data spaces. This can be a fast and convenient method to share data between otherwise isolated VMs. CP itself has to simulate the architectural differences from the underlying hardware (unsupported instructions).

[22]

3.4.5 Storage types, Paging Sub System

The z/VM operating system as such operates with three different types of storage (memory): main storage, expanded storage and paging space.

Main storage is chunk of central storage (real memory) made available for the z/VM operating system by the hardware (potentially virtual hardware). Program execution, data processing and I/O operations are performed here.

z/VM uses **expanded storage** as a fast paging device: A place where pages (4kb blocks of memory) can be moved, when main storage is full. Expanded storage is only addressable in whole pages, which in return can be moved efficiently between expanded and main storage. Earlier the expanded storage was dedicated hardware different from main storage. Today it merely a piece of ordinary central storage assigned to the logical partition as expanded storage.

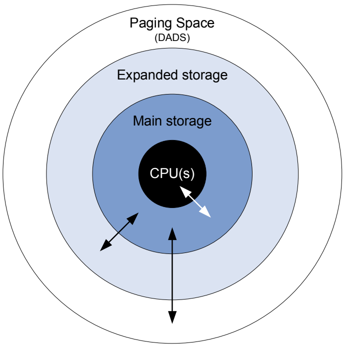


Figure 3-6: z/VM operates with three storage (memory) types. The processors can only work on data in main storage. Pages can only be moved between the different areas via main storage.

Actually z/VM is the only z/Architecture operating system, which uses expanded storage. This is feasible since it can host guests running in 31-bit addressing mode, which actually use it. Furthermore the VM paging sub system has over time been optimized to expanded storage and it therefore generally performs better when available.

Paging space is supplied by the **DASD paging sub system**, which can move pages to disk. This enables over-commitment of real memory: That is, too allocate more memory than actually available to the collection of virtual machines. The trick is to move rarely used memory segments of the virtual machines to disk and back when eventually called for – this without the guest ever noticing (except of speed).

[80] [59]

3.4.6 Service Virtual Machines

As indicated earlier, CP is "only" a hypervisor and not a fully fledged operating system. CP has no understanding of programs, processes or threads – basically all CP knows is how to run virtual machines. The consequence is that any extra functionality has to be implemented as virtual machines.

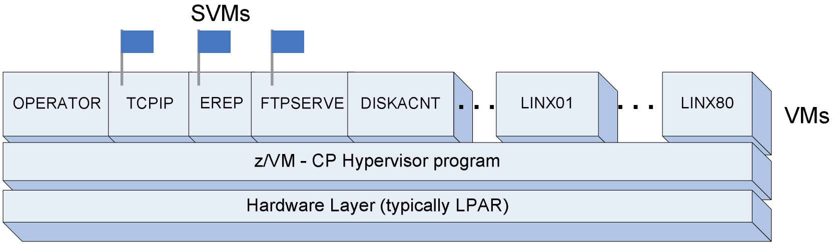


Figure 3-7: Services like a basic TCP/IP stack, FTP and error recording (EREP) are provided as *Service Virtual Machines*, which in fact are no different from other VMs.

General services relevant for some or all users/guests are therefore implemented as so-called **service virtual machines** (SVMs). Examples of

SVMs included in z/VM are: “RSCS” for remote spool device support; and “EREP” for error recording. A whole group of SVMs are related to network capabilities: First of all “TCPIP”, which provided a basic TCP/IP stack. Also many services like FTP, SMTP and virtual network switches are provided as service virtual machines.

Actually SVMs are completely identical to all other virtual machines on the system, they just run different software. SVMs might have higher scheduling priority than other virtual machines on the system. This is done to provide fast processing of low levels services for other guests. But priority boost could be given any other VM. SVM users are normally declared with the “SVMstat” option, but this is merely a descriptive flag, which can be used by monitoring programs etc. to distinguish SVMs from “normal” virtual machines.

[66][28]

3.4.7 General devices terminology

It should be clear by now, that the Control Program (CP) assigns resources to virtual machines. The following lines up the methods and present the related terminology.

A **real device** is typically what the name indicates: An actual physical hardware device. But from a more general z/VM perspective, **real devices** are the devices, which are available by the hardware for the z/VM operating system and thereby for the virtual machines running within. (In the case where z/VM runs inside z/VM in a virtual machine, real devices are actually virtual devices in the outer z/VM.)

Within z/VM (from CP’s point of view) a real device is identified by its “real device number” often abbreviated “**rdev**”. Be aware that **RDEV also** denotes a “**real device control block**”, which is a piece of storage CP associates with a real device containing information about that device’s features and status. Normally CP detects (senses) hardware devices and creates these control blocks automatically.

All devices inside virtual machines (made available by CP) are **virtual devices.** This applies regardless of whether resources are shared between, or assigned to distinct virtual machines (dedicated). CP identifies virtual devices using **virtual device numbers (vdev),** which are 3-4 digit hexadecimal numbers. CP handles virtual devices using VDEV control blocks (storage areas holding device relevant information).

The term **dedicated device** is used from two perspectives. 1) Within virtual machines: For virtual a device, to which CP has exclusively allocated a real device. 2) From CP’s perspective: For a real I/O device that CP has allocated exclusively to a virtual machine.

The permanent I/O configuration of virtual machines is defined in the “user directory” (see section 0). Devices can be added, detached, or changed temporary using CP commands. A **Temporary device** is one that is automatically detached and disappears when the user log off (the virtual machine is “turned off”).

[31][27]

3.5 Basic maintenance and configuration

It is out of the scope for this thesis to explain how z/VM is configured, maintained, operated or administered. It is however appropriate to briefly introduce a few fundamental concepts related to these issues.

3.5.1 Service (maintenance)

If a z/VM software bug is met, it is reported to IBM creating a problem management record (**PMR**). If a fix is needed, IBM creates an authorized program analysis report (**APAR**) as formal method to track the problem. IBM releases a program temporary fix (**PTF**) to solve the problem (somewhat similar to Microsoft Windows hotfixes).

Regularly IBM releases certain “Recommended Service Upgrade” (**RSU**s) containing important PTFs to lift the “service level”. These are somewhat comparable to a “servicepack”. Certain cumulative RSUs, which provide service updates for all z/VM components, features and products, are referred to as **stacked RSU** [12].

Installing program updates (or new program) in z/VM is unfortunately not quite as easy as clicking the “Windows Update” or “up2date” icon within Microsoft Windows or Linux. z/VM does come with a installation/service tool, **VMSES/E** (see “z/VM VMSES/E Introduction and Reference” [18]), which can be used to installing, migrating, building, deleting, and servicing software on the system. VMSES/E also provides tools for managing system software.

Earlier it was a long procedure with multiple tasks to “service” the system (refer to “z/VM Service Guide” [17] for details). The procedure is now automated with the SERVICE and PUT2PROD commands, which makes service quite easy (see “z/VM: Guide for Automated Installation and Service” [14]).

Program or SVM specific maintenance information, for example for “z/VM Performance Toolkit” and the TCPIP Service Virtual Machine, can be found in the so-called “**Program Directory” documents** available here: http://www.vm.ibm.com/progdir/.

3.5.2 Configuration

PARM disk

Three special minidisks are allocated as “PARM disks”: CMS formatted disk (exceptionally) readable by CP. z/VM uses one of these disk during “IPL” (when “booted”) to obtain information on the system definition. The second and third disk is mainly for backup purposes. These disks includes the **sys-**

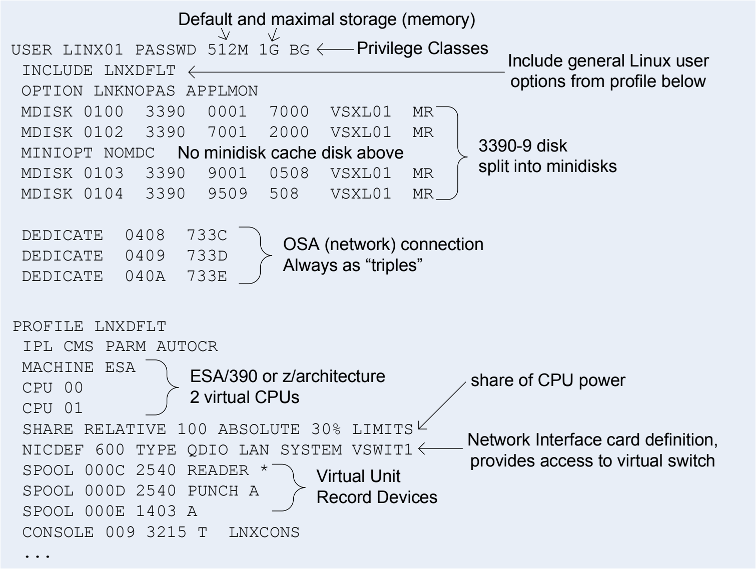
tem config file (see below), the **CP nucleus** (similar to the Linux kernel) with by default is called “CPLOAD MODULE”, and the logon screen logo configuration file. The disks are accessible within the virtual machine “MAINT” as virtual device CF1, CF2, and CF3.

System Config

The **system config** file on PARM disk is one of the most important configuration files. It can to some degree be compared with “config.sys” within MSDOS. It defines among other things the devices, which CP should bring online on start-up; it includes time zone settings; and it identifies the disks (DASDs) CP uses for spool, temporary disk, etc.

User Directory

The user directory is a flat “text file” owned by the MAINT user (vdev `2CC`). It contains all user (virtual machine) definitions: Their names, password, accessible devices, number of virtual CPUs, CPU share, the privilege class, and many other options. An example of how a user is defined is given in Figure 3-8. The following section (3.6) touches upon several of USER DIRECT statements, which are use to allocate virtual devices.



```
USER LINX01 PASSWD 512M 1G BG
 INCLUDE LNXDFLT
 OPTION LNKNOPAS APPLMON
 MDISK 0100  3390  0001  7000  VSXL01  MR
 MDISK 0102  3390  7001  2000  VSXL01  MR
 MINIOPT NOMDC
 MDISK 0103  3390  9001  0508  VSXL01  MR
 MDISK 0104  3390  9509  508   VSXL01  MR

 DEDICATE  0408  733C
 DEDICATE  0409  733D
 DEDICATE  040A  733E


PROFILE LNXDFLT
 IPL CMS PARM AUTOCR
 MACHINE ESA
 CPU 00
 CPU 01
 SHARE RELATIVE 100 ABSOLUTE 30% LIMITS
 NICDEF 600 TYPE QDIO LAN SYSTEM VSWIT1
 SPOOL 000C 2540 READER *
 SPOOL 000D 2540 PUNCH A
 SPOOL 000E 1403 A
 CONSOLE 009 3215 T  LNXCONS
 ...
```

Default and maximal storage (memory)
Privilege Classes
Include general Linux user options from profile below
No minidisk cache disk above
3390-9 disk split into minidisks
OSA (network) connection Always as “triples”
ESA/390 or z/architecture 2 virtual CPUs
share of CPU power
Network Interface card definition, provides access to virtual switch
Virtual Unit Record Devices

Figure 3-8: A section of the USER DIRECtory file. The user (virtual machine), LINX01, is an example of a Linux guest.

The `DIRECTXA` program is used to “compile” the `USER DIRECT` file into a file readable by CP. The user configuration is thereby activated. Any changes made to already running guest are not immediately effectuated but first applied on next user login.

3.6 Virtual devices: allocations, sharing

Any (virtual) machine can be abstracted (or reduced) into three basic resource types: processor/CPU power, storage (memory), and I/O devices (typically disks and network connectivity). The following sections describe common z/VM virtual resources; principles of sharing, and the statements in the USER DIRECT file (see section 3.5.2 above) used for their allocation.

3.6.1 Processors

The initial number of virtual processors within a virtual machine is given by the number of CPU statements within the user definition. Virtual processors can be defined as **dedicated** and thereby be assigned to a real CPU. It is thereby taken out of the pool of processor, which CP shares between all the virtual machines. Processors should only be dedicated to guests, which operate with CPU utilization greater than 90% for sustained periods of time [30]. The following example shows how to allocate two virtual CPUs with addresses `00` and `01`, one of which dedicated to a real CPU:

```
CPU 00
CPU 01 DEDicate
```

The number of virtual processors can (to some degree) be changed dynamically when a virtual machine is running. The maximum number of virtual processors, which can be activated from within the virtual machine, can be given as parameter to the machine statement, which otherwise only defines the virtual machine architecture. To set a maximum of 3 virtual CPUs for a virtual machine in ESA architecture mode:

```
MACHINE ESA 3
```

To illustrate how this feature can be applied, the following example shows the command used to activate a processor dynamically (if the maximum limit allows it) from a running Linux guest (replace “2” with an appropriate value):

```
 echo "1" > /sys/devices/system/cpu/cpu2/online
```

The number of virtual processors within a VM should never exceed the number of real processors made available for dispatching by CP.

3.6.1.1 Processor shares

Being able to prioritise users (virtual machines) and to control their consumption of processor resources is an important quality of a hypervisor like CP. This can be an important factor in order to allow a mixed environment, where production virtual machines run without being influenced by less important test VMs.

CP adjusts the processor dispatching according to overall goals, which are specified with `SHARE` statements in the `USER DIRECT` file (and equivalent

CP commands). Depending on the workload distribution and steadiness, CP can be expected to satisfy overall goals within a minute.

The syntax diagram for the `SHARE` statement is given in Figure 3-9: The first value (*y%* or *z*) denotes the "normal share" or "target minimum share". This is the amount of resources, which CP attempts to provide to a virtual machine as a minimum (if the VM can use of it, "not idle" that is).

The second value (*a%* or *b*) is the so-called "Maximum share". CP makes an effort to limit a virtual machine from using more than this amount of processor resources. If the maximum share is not specified the minimum share is also the maximum. The max limit can be specified in three ways:

- `LIMITHARD`: The limit is enforced, the VM does not receive more than specified; the VM is "capped".
- `LIMITSOFT`: The limit only enforced, when other users can use the resources.
- `NOLimit`: The user is not limited (this is the default; maximum share is not explicitly stated).

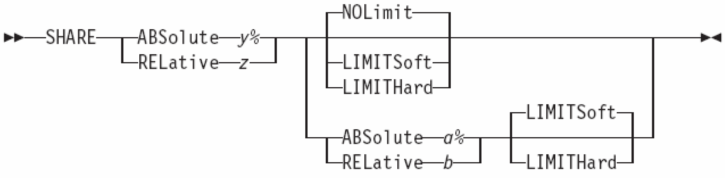


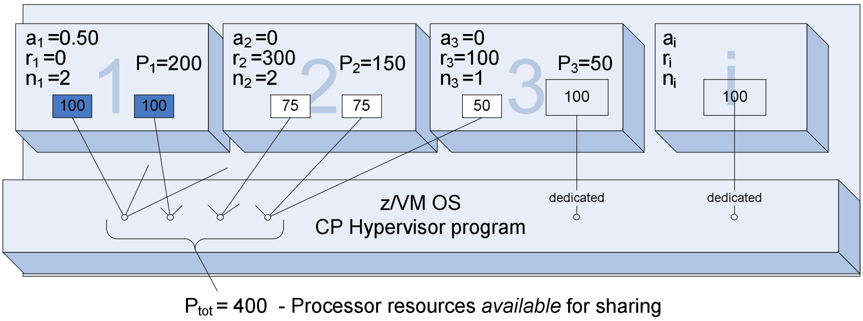
Figure 3-9: Syntax diagram for USER DIRECTory "SHARE" statement, which is used to control the percentage of processor recourses a user (VM) receives. [28]

The share values can be specified either as absolute or relative. An **absolute** value denotes an absolute percentage (0.1% to 100%) of the available processor resources (dedicated processors excluded). If the sum of absolute shares exceeds 99%, CP normalizes the values internally: 99% of the processor resources are proportionally distributed to the guests according to their absolute share value.

The **relative** share value is used to allocate the CPU resources not occupied using by absolute shares. That is minimum 1% of the available CPU resources but typically most of the resources. A relative share is given as an integer from 1 to 10000. CP assigns processor resources proportionally with respect to other virtual machines with relative share: The sum of all relative shares is calculated and the individual virtual machines are assigned their respective fraction of available resources.

An example is given in Figure 3-10. The "processing power" of the individual virtual processors, which is shown in the example, is given under the following assumption: All virtual machines are fully loaded and able to occupy all virtual processor with active processes or threads.

The concept of **processing power** is a fictive concept, which express the number of machine cycles a real processor is capable of performing in a unit time normalized too 100. This is in accordance with general mainframe perception, where a real processor is assigned the processing power of 100 (often 100 percent). The four sharable processors in the example yield a total of P_{tot}=400. The processing power assigned to individual virtual processors can thereby been seen as the percentage of what a real processor could do. The processing power of a virtual processor newer exceeds 100, since a virtual processor never operate faster than its real counterparts.



$$P_1 = 400 \cdot 0.50 = 200$$
$$P_2 = \tfrac{300}{100+300} \cdot (1-0.50) \cdot 400 = 150$$
$$P_3 = \tfrac{100}{100+300} \cdot (1-0.50) \cdot 400 = 50$$

Figure 3-10: Example of processor resources sharing. The four processors available for sharing by CP have a total computing power of 400 units. VM1 is assigned two virtual processors and an absolute share (a_1) of 50%. VM2 is assigned two virtual processors and a relative share (r_2) of 300. Finally VM3 is assigned a relative share (r_3) of 100 and given 2 virtual processors; one of which is dedicated and therefore kept out of the calculation. The computing power of the individual virtual machines, P_i, (dedicated processor excluded) is given in the figure. The computing power of the individual virtual processors is given inside the representing boxes.

As indicated, CP distributes the guest's share of computing power, P_i, between the virtual processors within the guest. If the second guest in the example only uses one of its CPUs, it will only be weighted 150 (half of 300) to 250 (sum of relative shares for active virtual processors) when the remaining the resources for processors with relative share is distributed. It appears that it can be quite tricky to foresee the distribution of processor resources in a more complex (realistic) situation. The following section provides a relatively simple method, which yields a processing power distribution comparable to measurable values.

[30][28]

Calculation of Processing Power distribution

This section presents the use of absolute and relative shares from a more formal point of view. The processing power of individual virtual machines and individual virtual processor are determined. Share limits (LIMITSOFT and LIMITHARD) are not considered. Similarly, the model does not differentiate between normal shares and maximum shares. The results can be expected to apply, when considering coarse-grained units of time like 1 minute. And finally, dedicated processors are not incorporated.

The general system and user configuration is given by the following symbols, which are shown in their respective ranges. Also refer to Figure 3-11.

$1 \le n_{real}$	Number of real processors available for sharing.
$P_{tot} = 100 \cdot n_{real}$	The total amount of processing power available.
$0.01 \le a_{\max} \le 1$	The largest fraction of P_{tot} allowed abs. shares ($a_{\max}$ =99%)
$0.01 \le a_i \le 1$	Absolute share value for guest *i*.
$1 \le r_i \le 10000$	Relative share value for guest *i*.
$1 \le n_i \le 1$	Number of virtual processors in guest *i*.

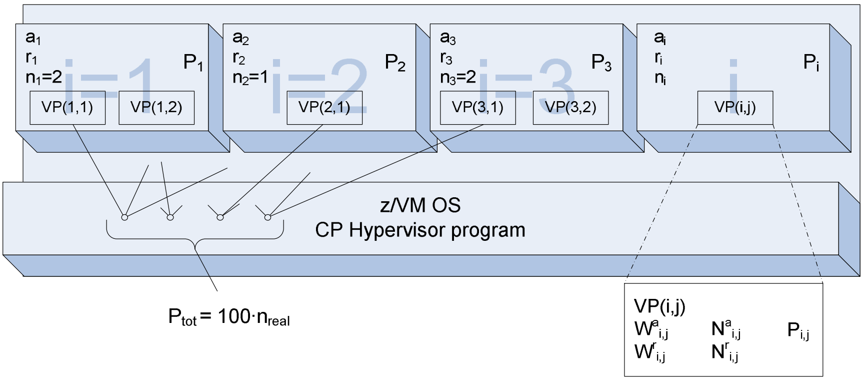


Figure 3-11: The conceptual z/VM system and user configuration used in the model. The illustrated example includes four virtual machines, i={1,2,3,4}, and four real processors for sharing, n_{real}=4.

Any *virtual processor* has a static share or “weight” according to the actual user configuration and the defined virtual CPUs. This weight can be calculated by dividing the share of the individual virtual machine evenly between the virtual processors inside it. Absolute and relative weights, W^a and W^r, are kept separately; for virtual processor *j* in virtual machine *i*:

$$W^a{}_{i,j} = \frac{a_i}{n_i}$$

$$W^r{}_{i,j} = \frac{r_i}{n_i}$$

Now, CP prioritises the *active* virtual processors according to their mutual weight: The priorities are given based on the individual processor's share with respect to the shares of all currently active processors. This influences how often they are dispatched and in the end how much processing power they receive. CP considers all virtual processors in the dispatch list (see section 3.7.2 p. 48) as "active".

CP does, in other words, not directly calculate *the processing power*, which a guest should receive. This is merely results of the dispatching priorities. In other to calculate these values it is therefore necessary to mimic CP's behaviour (to some extend).

First a list, *A*, of the *a*ctive virtual processors is maintained. In this simplified model, all processors, which have something to do in the considered unit of time, are considered active. The sums of the processor weights for the corresponding virtual processors are calculated continuously (as processors are added or removed from the list):

$$A = \textit{the set of active virtual processors } VP(i,j)$$

$$S_a = \sum_{VP(i,j)\in A} W^a{}_{i,j}$$

$$S_r = \sum_{VP(i,j)\in A} W^r{}_{i,j}$$

The weights of the *currently active* virtual processors are normalized to 1, according to the practise of allocation of absolute and relative shares. If the sum of absolute weights exceeds a$_{\max}$ (99% of total processor capacity) the absolute weights are normalised to the sum of a$_{\max}$. The normalized weight, N, for processor *j* in guest *i* is calculated like this:

$$N_{i,j} = N^a{}_{i,j} + N^r{}_{i,j}$$

$$N^a{}_{i,j} = \begin{cases} W^a{}_{i,j} & \textit{for } S_a < a_{\max} \\ W^r{}_{i,j} \cdot \frac{a_{\max}}{S_a} & \textit{for } S_a \geq a_{\max} \end{cases}$$

$$N^r{}_{i,j} = \begin{cases} \frac{W^r{}_{i,j}}{S_r} \cdot (1 - S_a) & \textit{for } S_a < a_{\max} \\ \frac{W^r{}_{i,j}}{S_r} \cdot (1 - a_{\max}) & \textit{for } S_a \geq a_{\max} \end{cases}$$

If the normalized weights alone were used to proportionally distribute the total amount of processing power, P_{tot}, between the virtual processors, some processor might receive more power than a real processor actually has. This is not acceptable. Another issue arises since an "active" virtual processor, might not be able to utilize a real processor for a complete unit of time. But since it still is in the dispatch list, possibly with a very high priority, it can still influence the priority of the other virtual processors considerable.

These issues are handled using an iterative approach and by introducing the concept of load. The **load**, *$L_{i,j}$*, denotes the actual amount of processing power a virtual processor *j* of guest *i* actually uses. In other words: the number of "normalized machine cycles" it has tasks to make use of in the considered unit of time. Since a virtual processor cannot process more than a real processor the load, *L*, it limited to the processing power of a real processor:

$$L_{i,j} \leq 100$$

The pseudo code in Listing 3-1 calculates the *processing power* assigned to the individual virtual processors in a particular unit of time. The virtual processors are processed one at a time in a loop. A couple of variables are introduced to support the calculations:

The variable ***remainN*** contains the sum of the **remain**ing **N**ormalized weights for the processors yet to be processed in the loop. In the beginning where no processors are processed, the value of *remainN* is 1.

The variable ***unexpl*** is used to hold the amount of "**unexpl**oited processing power", which should be divided between the remaining virtual processors. The amount is increased when a virtual processor cannot utilize the processing power is should be allocated according to the normalized weight. This can happen when the load, *L*, is too small or when the weight would give raise to more processing power than a single processor can supply (instated by *L*≤100).

Having computed the processing power of the individual virtual processors it is possible to find the processing power of the individual virtual machines, by summarizing the virtual processors individual contributions.

$$P_i = \sum_{1 \leq j \leq n_i} P_{i,j}$$

The calculations given above assume that the value of P_{tot} is constant. Actually the value might vary itself. This can happen when z/VM is running inside z/VM in a virtual machine (or then z/VM runs in a Logical Partition without dedicated processors).

In principle it should be possible to apply the calculation above twice: First in the inner z/VM assuming full resource availability to calculate how much processing power it can actually consume: the sum of *P_i* values. Next the processing power distribution in the outer z/VM can be determined using the data from the first calculation as load, *L*. Finally the distribution results from the first calculation have to be proportionally scaled down to fit the resources actually made available by the outer z/VM.

```
Pseudo procedure for P_{i,j} calculation
Initialize remainN to 1
Initialize unexpl to 0
Sort list of active processors, A, according to the processors normalized weight, N_{i,j}, with the largest values first.
For each actual virtual processor, VP(i,j), in the sorted list:
    Find the proportional share of total processing power according to normalized weight:
        $P_i^{\mathrm{Prop}} = P_{tot} \cdot N_{i,j}$
    Find share of unexploited processing power; proportionally divided between the remaining virtual processors according to their weights:
        $P_i^{Excess} = \mathrm{unexpl} \cdot \frac{N_{i,j}}{\mathrm{remainN}}$
    Find the actual processing power of the virtual processor, limiting the value to the actual load or the capacity of a real processor.
        $P_{i,j} = \min\left( P_{\mathrm{i}}^{\mathrm{Prop}} + P_i^{Excess} \,;\, L_{i,j} \right)$
    Update the value of "unexploited processing power" by applying the difference between the actually assigned value (P_{i,j}) and the value which should have be applied according to normalized weights (P_i^{Prop}).
        $\mathrm{unexpl} := \mathrm{unexpl} - \left( P_{i,j} - P_i^{\mathrm{Prop}} \right)$
    Update the sum of remaining normalized weights by subtracting the normalized weight of the actual virtual processor:
        $\mathrm{remainN} := \mathrm{remainN} - N_{i,j}$
```

Listing 3-1: Pseudo code to calculate the processing power, $P_{i,j}$, of the individual virtual processors *VP(i,j)*.

3.6.2 Storage (memory)

The amount of storage made available to a virtual machine is defined in the very first line of a user definition within the USER DIRECT file:

```
USER username password 256M 512M G
```

Two values can be given: The default size of the VM's primary address space and a maximum value. As in the case with the processors, it is possibly to change the storage size (within defined limits) from "inside the VM". This can be done with the following CP command (here changing the size to 512MB):

```
#CP DEFINE STORAGE 512M
```

This command is definitely not practical for running Linux guests, since it also clears the storage area and resets the virtual machine (like a "hard reset"). It can on the other hand be quite useful when running CMS, which better tolerates a reset and is easily restarted. The MAINT user is fine example: Under normal everyday conditions the MAINT user allocates 128MB, which is more than enough for running XEDIT and making changes to the USER DIRECT file. But in a service situation the storage area can easily be increased (up to 1000MB) and thereby provide plenty of space for building software.

The "storage sharing" and "storage over commitment" (paging) capabilities of the Paging SubSystem have been introduced earlier (3.4.5). This section will not dig deeper into the underlying mechanism of how and when to move pages to/from disk: Because in most situations, the Paging SubSystem handles this autonomously without any special configuration considerations to be done.

There are only a few special advanced commands, which can be used to tune the Paging PubSystem (e.g. `SET SRM STORBUF` and `SET SRM LDUBUF`). Their effect and use are however out of scope for this thesis. For further details please refer to the "z/VM Performance" book [30].

3.6.3 I/O Devices: DASD / Disks

CP can assign several types of Direct Access Storage Devices (DASDs, introduced in section 2.2.8 on page 15) to virtual machines.

Dedicated DASD is real disk, which CP has dedicated to a virtual machine. The guest operating system has full control over the device. A disk given by its "real device number" (rdev) or by its volume id (volid) can be assigned to virtual device number (vdev) using one of the following statement in the `USER DIRECT` file:

```
DEDICATE vdev rdev
DEDICATE vdev volid
```

3.6.3.1 Minidisks

It is often useful to divide a DASD device into smaller virtual disks. In z/VM such virtual disks are called **minidisks** and there exists three basic types of them: Permanent, temporary, and virtual disk in storage.

Permanent minidisks (often simply referred to as minidisks) are somewhat comparable to hard disk partitions (often created by "fdisk") on PCs. Both ECKD and FBA DASD can be divided into minidisks. They are part of a DASD device controlled by CP (they are attached to the system). The guest operating system manages a minidisk as any other DASDs, but the actual I/O operations are performed by CP, which transform the virtual addresses (cylinders, tracks or FBA block numbers) to their real counterparts. A (ECKD) minidisk definition statement in `USER DIRECT` could look like this:

```
MDISK 0103  3390  6677  3338  VSXD02  MR
```

The statement defines a virtual disk of type `3390` with virtual device number 0103. The disk is created on the real DASD device with volume serial number (volid) `VSXD02` beginning on cylinder number `6677`. The minidisk occupies `3338` cylinders and "multiple-write" access mode ("MR").

It can be tricky to define minidisk without making errors. z/VM provides a useful tool called `diskmap`, which can be used to check the user configuration for minidisk overlaps and gaps.

Minidisks are often shared between virtual machines for example to provide access to common tools and programs. The following statement shows how to provide access to a minidisk defined in another user called "TCPMAINT". In this case the same virtual device number (592) is used both virtual machines, but is not necessarily the case:

```
LINK TCPMAINT 592 592 RR
```

A **Full-pack minidisk** is a special kind of disk, which overlays all minidisks on a DASD and thereby provides a single point of access to the data on all the minidisks.

Temporary disks (T-disks) are the second type of minidisk. T-disks are "destroyed" when the user logs off. The space is taken from a pool of temporary disk space when the user logs on and returned to the pool when the user logs off. A T-disk is also defined using the MDISK user directory statement.

VDISK or "**virtual disks in storage"** is the third and final type of "minidisks". These disks are also temporary disks but they reside in main storage (memory) instead of on disk: A concept similar to "ramdisks" used in MSDOS and Linux. This makes them faster than other disk types since the I/O operations are eliminated. VDISK uses FBA (Fixed Block Architecture) as access scheme, because continuously numbered blocks match the memory access scheme much better than addressing by cylinders and tracks (ECKD). VDISKs are often used in relation to Linux guests as fast swap disks.

[31][28][59]

3.6.3.2 Emulation of "SCSI via fibre" as FBA DASD

z/VM provides native support for SCSI disks attached via FCP (Fibre Channel Protocol), as briefly mentioned in the mainframe presentation in chapter 2. As a result both z/VM itself and guest operating systems can be installed on and operate on SAN disks (Storage Area Network).

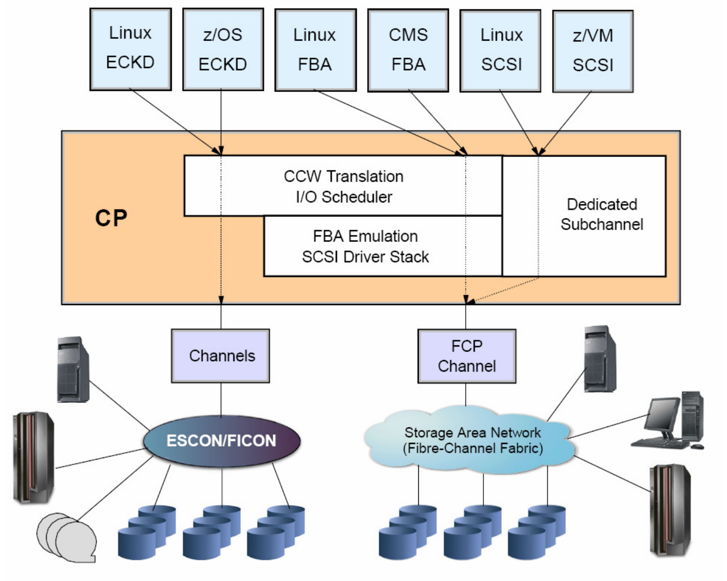


Figure 3-12: SCSI support; CP emulates SCSI disk into FBA DASD devices for use in Linux or CMS. Linux and z/VM guest can also access SCSI device directly. [59, p. 44]

Guest operating systems like Linux which have "driver support" for these SCSI devices can access them directly as dedicated devices. The second access methods is a good example of CP's device emulation capabilities: CP emulates SCSI disks as "standard" FBA disks (actually 9336 model 20), which can be used by CMS or Linux like any FBA disk e.g. VDISKs (see Figure 3-12).

[59][60]

3.6.4 I/O Devices: Network connectivity

The networking capabilities of z/VM is wide-ranging and far beyond the scope of this thesis. The group of real/physical networking interfaces/adapters, which can be used in virtual machines, includes:

- Several types of OSAs (Open System Adapters),
  e.g. providing access to many Ethernet, token-ring and ATM networks.
- Channel-to-channel-adapters
  proving a mainframe point-to-point connection using channels.
- Common Link Access to Workstations (CLAW)
- HiperSockets (hardware provided virtual LANs)

To supplement the real devices made available by hardware, z/VM includes several virtual network technologies:

- Point-to-point connectivity (virtual channel-to channel adapters, CTCA, or the Inter-User Communication Vehicle, IUCV, facility).
- Guest LAN
- z/VM Virtual Switch (VSWITCH)
- Layer 2 LAN Switching.

The following introduces the use of OSA/HiperSockets and VSWITCH, which probably are the most commonly used network types used in Linux guests under z/VM. For further details refer to the “z/VM Connectivity” book [26] and chapter 4 “Networking Overview” in the IBM Redbook “Linux for IBM System z9 and IBM zSeries” [59].

3.6.4.1 Open System Adapter and hiperSocket connectivity

OSA connections (e.g. to a Gigabit Ethernet based LAN) and HiperSockets connections (to interval hardware provided virtual LANs) are almost alike, both being based on QDIO (remember section 2.2.9 p. 16).

From operating system perspective both interface types appear as a whole range of I/O devices: a range real devices in z/VM that is. It requires a total of three consecutive devices per connection: The first device (an even numbered) is used for read control, the next for write control, and the last for data (see Figure 3-13).

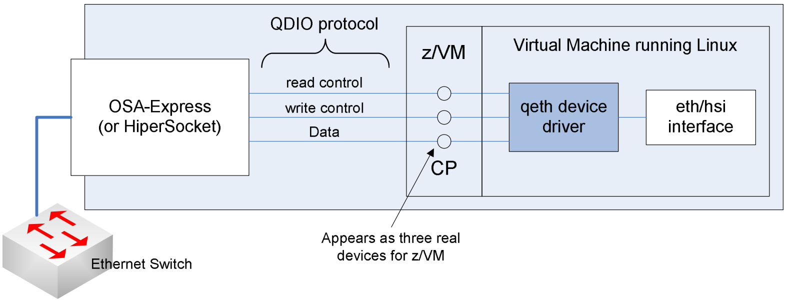


Figure 3-13: QDIO based networking: OSA (and HyperSockets) interfaces appear as three I/O devices (z/VM RDEVs) per connection.

To give a virtual machine access to OSA or HiperSockets simply dedicate a set of three devices in the `USER DIRECT` file. For example to dedicate real devices 733C-733E to vdev 0408-040A:

```
DEDICATE 0408 733C
DEDICATE 0409 733D
DEDICATE 040A 733E
```

[59]

3.6.4.2 Virtual Switch (VSWITCH)

A virtual switch (VSWITCH) is virtual device provided by z/VM. VSWITCH'es can operate either at Layer 3 (network layer in OSI model) using IP addressing; or at Layer 2 (data link layer in the OSI model) working at MAC address level. The technology also supports IEEE 802.1Q VLANs. It can be used to connect several virtual machines to an external LAN provided via an OSA interface. A switch can actually also run in disconnected mode (without an associated OSA port) in order to interconnect virtual machines though a virtual switch almost equivalent to a physical one (Layer 2).

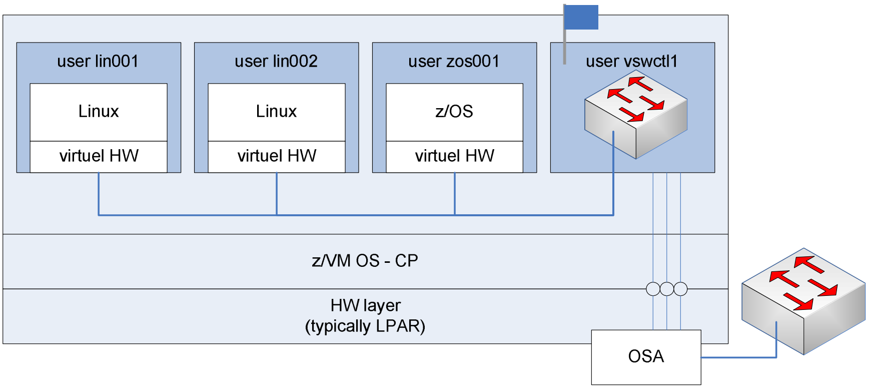


Figure 3-14: VSWITCH setup (simplified). Service Virtual Machines (SVMs) are used to provide a virtual switched network, possibly connected to an external LAN, for z/VM guests. The switch is typically connected to an external LAN via an OSA interface (or to a internal hardware based LAN via HiperSockets).

A virtual switch in z/VM requires a SVM (Service Virtual Machines) to function as "controller" and to enable the connection to the OSA interface (Figure 3-14). Typically two SVMs are used for backup and isolation reasons.

The configuration of SVMs and virtual switches in generally are not covered here, with the exception of the following two notes.

A VSWITCH is statically defined; and dedicated to one (or more failover) OSA interface(s) in the `SYSTEM CONFIG` file. This is also where virtual machines (users) are granted access to a virtual switch:

```
DEFINE VSWITCH vswdemo0 RDEV 733C 733D 733E VLAN 1
MODIFY VSWITCH vswdemo0 GRANT LIN001
MODIFY VSWITCH vswdemo0 GRANT LIN002
```

The individual users are connected to the switch by including a "network interface card definition" statement in their `USER DIRECT`ory definition. For virtual device number 600 connected to virtual switch VSWDEMO0:

```
NICDEF 600 TYPE QDIO LAN SYSTEM VSWDEMO0
```

[59]

3.6.5 I/O Devices: Unit Record Devices (spooling)

When introducing virtual devices in z/VM, the concept of Unit Record Devices should not be overlooked. Generally Unit Record Devices embrace printer, punch, and reader devices – and their virtual equivalents within z/VM. It is, in other words, "remains" from the days where data/programs entered the system though punch cards and output was written directly to the printer.

In order to enable sharing of these real unit record devices their virtual equivalents use spooling. **Print spooling** is a known technique used on most platforms to enqueue print jobs and to avoid the application blocks until the print job has finished. In z/VM virtual readers and puncher also use spooling.

Virtual unit record devices use a **spool file system** handled by CP. One or more DASD volumes are assigned to CP for this specific purpose. Within the spool file system is **spool files**, which contain a collection of data together with device control instructions for processing on a unit record device. It is the processing of these files created by, or intended for, virtual readers, punches, and printers, which is called "**spooling**".

It is possible to send spool files from one virtual device to another, from your virtual machine to another, and to real devices, using CP and CMS commands. But the importance and practical use of Unit Record Devices are highly diminished in comparison to earlier. The devices are nevertheless an integral part of z/VM and they still serve a few practical purposes. These include console logging and facilities for storing dumps of erroneous virtual machines.
[47][31]

3.7 Virtualization details for z/VM

3.7.1 Hypervisor calls, Diagnose instructions

As briefly mentioned earlier, z/VM supports **paravirtualization** and thereby provides special **hypervisor calls** to be used by the guest operating system. An operation system, which is aware of it running in a virtual environment, can use these hypervisor calls to "communicate" with CP.

The interface provided by CP is the so-called "**DIAGNOSE**" instruction. The instruction contains a code portion (DDD), which specifies the actual service to be performed, please refer to Figure 3-15. Then a diag instruction is issued, the control is handed to CP. CP examines the code, performs the matching operation and returns control to the virtual machine.

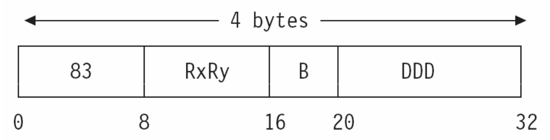


Figure 3-15: The machine language format of the DIAGNOSE instruction. The first part, “83”, is the machine language instruction code (an assembler mnemonic does not exist). RxRy gives the general registers, which hold operand values or storage addresses to such. B is a displacement base register for DDD, which is the actual DIAGNOSE code value specifying the service to be performed. (Graphic from [29, p. 3])

IBM have implemented a variety of DIAGNOSE code into z/VM. Only a few examples are given here:

- Code X’44’ – “Voluntary Time Slice End”:
  Informs the scheduler that the virtual CPU cannot make use of the remaining time slice, because a spin lock loop exists.
- Code X'A4' – “Synchronous I/O”:
  Used to perform synchronous I/O operations on CMS formatted DASD letting CP construct the appropriate channel program.
- Code X'288' –“Control Virtual Machine Time Bomb”:
  basically a watchdog timer for virtual machines.

Some DIAGNOSE codes can only be issued by authorized users within special privilege classes. Other codes require a special architectural mode or storage access mode. It should be noticed, that DIAGNOSE instruction also exist outside z/VM. When issued outside a VM, the instruction makes the running processor perform build-in diagnostic or other model-dependent functions. These operations can be potentially cause fatal mal-functions. [11] Operating systems (other than CMS) should therefore prevent execution of DIAGNOSE instructions in “real machines”. The “Store CPU ID” instruction can be used to determine the actual environment.
[29]

3.7.2 Processor Scheduling

The scheduling and dispatching routines in z/VM are quite complex. The following introduces the main elements from a simplified point of view. Chapter 2 in the “z/VM Performance” book [30] is recommended for a more detailed description.

The main task of the z/VM scheduler is to make the virtual machines appear to be running concurrently by assigning them “time slices” of the real processors. The scheduling is based on the virtual machines’ actual need for, and the availability of, processor cycles, real storage, and paging space.

CP cycles the active virtual machines (the ones logged on) through three lists in order to determine processor dispatching order and priorities. The virtual machines move between these lists according to their current work engagement; see Figure 3-16.

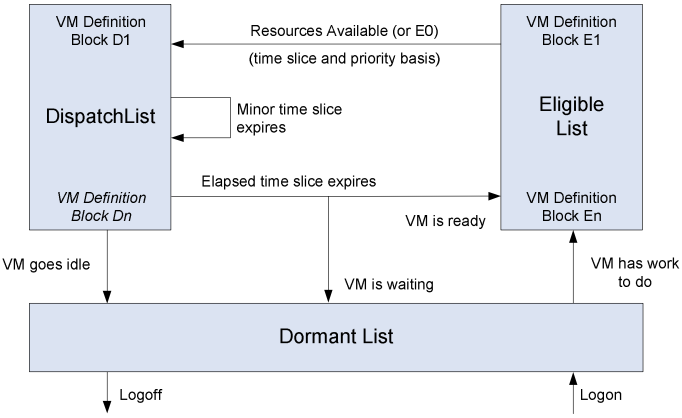


Figure 3-16: The z/VM scheduler/dispatching lists; and flow of virtual machines between them.

The **dormant list** contains VMs, which have no immediate task to perform. They can be idle (e.g. awaiting user interaction), in an enabled wait state (e.g. waiting for a timer interrupt), or they can be waiting for the completion of a long operation (e.g. waiting for a long page-in operation to finish).

The **eligible list** comprises virtual machines, which are waiting for system resources. These VMs are priorities into four **transaction classes:**

E0: Virtual machines, which do not wait in the eligible list for resources to become available, but have a special status, e.g. the quick dispatch option.

E1: Virtual machines expected to have short transactions (in z/VM a transaction is a basic “unit” of work). They have just started a transaction.

E2: Virtual machines with expected medium length transactions. They have returned from an “E1 stay” in dispatch list on elapsed-time-slice-end, without finishing their work.

E3): VMs execution long-running transaction, having returned from at least an E1 and an E2 stay in the dispatch list.

The final list, the **dispatch list**, contains the virtual machines, which are ready to run and compete for processor time. The dispatch list also includes virtual machines whose waits are expected to be short. When contention for processor time exists, the share settings (ABS or REL SHARE) are used to control the virtual machines’ priority within the dispatch list. When virtual machines are moved from the eligible list to the dispatch list, they basically retain their transaction class E0-E3 although “renamed” Q0-Q3 respectively.

These lists enable CP to favour "interactive" virtual machines over non-interactive (batch oriented) virtual machines, in order to provide good response times.

[30] [59]

3.7.3 Running z/VM in z/VM

Since the virtual machines created by z/VM control program (CP) fully resembles real machines, they also are capable of running the z/VM operating system itself. The first z/VM running directly on hardware (often in LPAR) is called as a "first level" system and it runs first level guests. A z/VM running in one of these guests is a "second level" system running second level guests (see Figure 3-17).

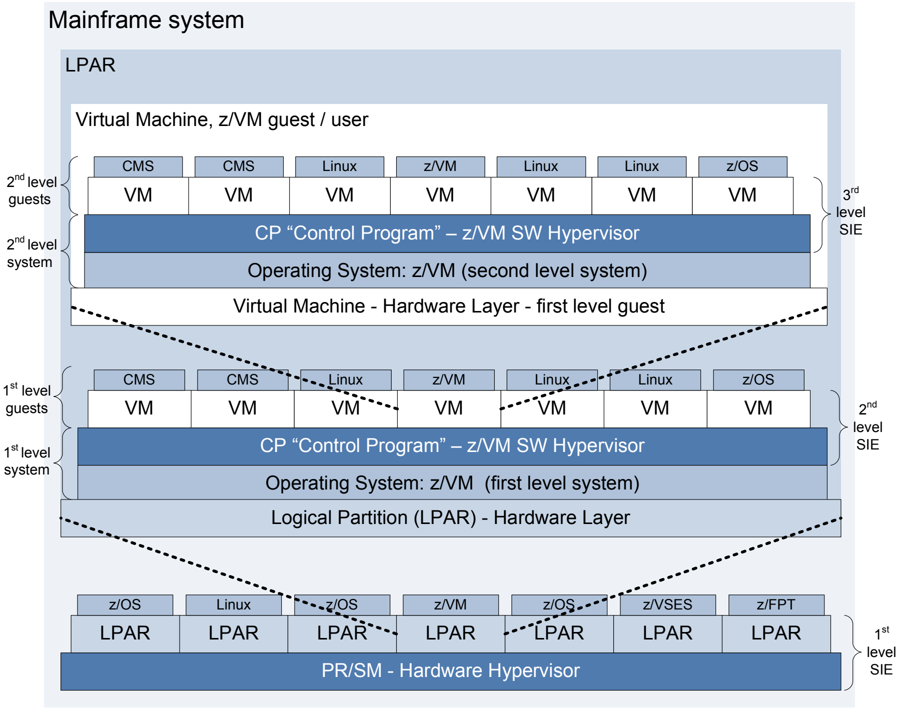

Mainframe system
LPAR
Virtual Machine, z/VM guest / user
2nd level guests
CMS CMS Linux z/VM Linux Linux z/OS
VM VM VM VM VM VM VM
3rd level SIE
2nd level system
CP "Control Program" – z/VM SW Hypervisor
Operating System: z/VM (second level system)
Virtual Machine - Hardware Layer - first level guest
1st level guests
CMS CMS Linux z/VM Linux Linux z/OS
VM VM VM VM VM VM VM
2nd level SIE
1st level system
CP "Control Program" – z/VM SW Hypervisor
Operating System: z/VM (first level system)
Logical Partition (LPAR) - Hardware Layer
z/OS Linux z/OS z/VM z/OS z/VSES z/FPT
LPAR LPAR LPAR LPAR LPAR LPAR LPAR
1st level SIE
PR/SM - Hardware Hypervisor

Figure 3-17: z/VM is capable of running inside a z/VM virtual machines.

It is possible to run z/VM as even higher level systems (third, fourth, ... in "Russian doll" style) but it is rarely very valuable. Two levels can however be useful in several cases including the following (especially the latter case has proven very useful in relation to this particular project).

- To test a new version of the z/VM OS itself
- To test (shared CMS) application programs
- To test new service levels ("patches")
- To test new service and maintenance procedures

- To *train* personal and provide an environment for experiments.

All of the above can be achieved without influencing the 1st level production z/VM in which the 2nd level test z/VM runs. The 2nd level system will naturally take up some resources but the SHARE setting can be used to assign the system a relevant priority in relation other virtual machines in the 1st level system. The total isolation provided by the virtual machine otherwise ensures that any disastrous experiment only affects the test system.

3.7.4 Hardware supported virtualization

Interpretive Execution

As briefly mentioned earlier, the mainframe hardware has direct support for virtualization. CP uses the same mechanisms, as the HW hypervisor PR/SM uses to dispatch shared processors between Logical Partitions: namely SIE (**Start Interpretive Execution**).

CP calls the **SIE** instruction to place the processor in **interpretive-execution mode**: to work on behalf of the virtual machine. In order to establish the correct virtual environment, SIE takes an operand, the so-called "**state description**", which contains the state of the virtual processor (its "context"). This includes the Program Status Word (PSW); general registers; and means for accessing the guest's Dynamic Address Translation (DAT) structures (region, segment and page tables). The state description also defines the reasons for exiting interpretive-execution mode; the reasons to "trap" back" to CP (the host program) that is.

In interpretive-execution mode the virtual processor can utilize all functions offered by the architecture with speed comparable to "native" execution (without virtualization or partitioning). The virtual processor run in "SIE mode" until an interception condition is raised (e.g. time slice expires or interrupt is received). When such a "SIE break" occurs, the state description (including PSW) is updated and the control returned to CP (or PR/SM)

Now since a first level z/VM system typically runs in a LPAR (especially since newer mainframes cannot run without Logical Partitioning) it give rise to SIE calls from at least two levels (see Figure 3-17). The mainframe hardware actually allows this using "Interpreted SIE".

Interpreted SIE is hardware function, which was introduced to let VM/ESA (the 1990 version of z/VM) run smoothly as a guest of itself. Interpreted SIE allows a machine already in interpretive-execution mode to call SIE and institute yet another instance of interpretive-execution mode – still under HW control [65]. The mainframe platform is actually the only technology on the market, which provides *two* levels of hardware support for virtualization [49]. On older mainframes capable of running in basic mode (without LPAR) z/VM could run in two levels, in both of which most features perform without performance loss.

The hardware is, however, only capable of handling two levels of SIE. Since PR/SM and the logical partitioning uses the first SIE level, the requirement for running z/VM in z/VM in LPAR like depictured in Figure 3-17 is three levels of SIE. The 3rd level SIE is therefore emulated by the first level z/VM. Actually third SIE-level and above is said to "pancake" down to SIE level 2 [49]. This emulation is expensive and results in performance degradation.

[58] [65] [50]

QDIO assist

IBM continuously implements new technology to improve virtualization performance. The z/VM version 4.4 on zSeries hardware allowed adapter interrupts from HiperSockets, FCP based devices and OSAs (Open System Adapters) to be passed directly to guest operating systems, which exploited the QDIO communication mode (Queued Direct I/O, mentioned in 2.2.9 page 16).

Later "hardware assists" include "QDIO Enhanced Buffer-State Management" and "Host Page-Management Assist". These technologies allow a virtual machine to initiate QDIO operations directly to an appropriate channel, without interception by CP.

From Linux guest perspective, the effect of the above is that QDIO based devices (including network interfaces, OSA, HiperSockets, and FCP connected iSCSI disks) can be used without interception by the CP. This naturally gives a performance improvement. Unfortunately these new virtualization technologies are available only to first-level z/VM guests. Second level guest cannot make any use of them.

[22]

4. LINUX ON ZSERIES, AS Z/VM GUEST

Running Linux on the mainframe is almost in every aspect as running Linux any other platform. An average (non GUI) end user would practically never know the difference. Server administration, on the other hand, is a little different due to the architectural differences. The general methods (editing configuration files in /etc, utilizing the sysfs file system, or using a configuration tool like SUSE YaST) are still the same.

This chapter introduces some of the most important issues, which set apart mainframe Linux from other platforms. The reader is expected to be familiar with Linux beforehand.

The chapter is restricted to general issues of kernel 2.6 based systems running in 64bit mode exploiting the z/Architecture. The 2.6 kernel includes many improvements compared to 2.4. The 64bit mode is generally recommended offering greater memory addressability and providing greater flexibility [59, p. 10]. Systems running in 31bit mode or based on kernel 2.4 do still exist but these are mainly running on older hardware or in long-running but well-functioning setups.

4.1 Presentation

4.1.1 History

The first Linux Mainframe project was called “Bigfoot” and was founded by a visionary (almost clairvoyant) Linus Vepstas in 1998. He basically believed that Linux could be the way for mainframe vendors to regain lost mainframe costumers to and acquire new ones – now almost ten years later he has been proven right. His own Bigfoot project was however abandoned in 2000 after IBM had announced a competing project. [75] [76]

IBM started the development of the code required to run Linux on S/390 architecture in 1999. IBM’s focused on adapting the 2.4 kernel to the zSeries platform, but their work was not particular well integrated with the rest of the community. As an example their patch sets were often very large and therefore not included in the mainline kernel, since the kernel maintainers usually prefer smaller changes to allow manageable reviews. They had little influence on the Linux development process in general and one of the most obvious advantages with open source, the peer review process, was not exploited. [52]. IBM furthermore keep some issues secret e.g. by only providing binary device driver to OSA devices. [76]

With Linux 2.6 kernel development IBM became committed to Linux and open source. Drivers are released as open source and IBM has participated actively in the community kernel development process. This has allowed them to oppose to changes, which influences “zLinux” negatively. IBM has developed solutions, vital for the z/Architecture, which also have benefited other platforms and the kernel in general (e.g. in regard to memory management and a CPU hot plug feature).

[52]

4.1.2 Architecture reference names

The Linux ports for the mainframe are called "s390" and "s390x". These are the names used to reference the 31bit and the 64bit mainframe architecture respectively. In other words: The ESA/390 architecture is denoted "s390" and the z/Architecture is denoted "s390x".

4.1.3 Distributions

There are several Linux distributions for the mainframe. Two of these are commercial and enterprise class supported distributions:

- Novell / SUSE Linux Enterprise Server (SLES), latest version: 10 (SP1)
  http://www.novell.com/products/server/
- Red Hat Enterprise Linux (RHEL) , latest version: 5
  http://www.redhat.com/rhel/server/mainframe/

Several other free alternatives exist but some of them have not been updated for year. The following should be the remaining active projects:

- CentOS
  http://www.centos.org/
- Debian (31bit only)
  http://www.debian.org/ports/s390/
- Slack/390
  http://www.slack390.org/

CentOS is actually also an enterprise quality distribution, since it basically is a "de-branded" version of RHEL. This is completely legal and in full agreement with the open source licenses. It is intended to provide an enterprise class OS without the related costs and support.

Generally there seem to be a common agreement that SUSE Linux Enterprise Server is mostly widely used distribution on the mainframe. [53] All activities in this particular project are performed in a SLES9 SP3 environment.

4.2 Devices and sysfs

With kernel 2.6 the "system file system" (sysfs) provides a unified hierarchical view of the hardware. Devices are represented as a subdirectory containing files or "attributes". These attributes can be used to access information about the device or used to control them (for example to enable/disable them or set them in a specific mode). The directory structure itself shows how devices are interrelated or connected e.g. via a bus.

In Linux for zSeries directory structure provides information on the device type, device node name, and subchannel numbers. Hardware devices are first and foremost listed in /sys/devices/ccs0/ (see Figure 4-1). In this directory all available channels (chp0.XX) and subchannel (0.0.xxxx) are given as sub directories. In the “subchannel subdirectories” yet another directory exists representing the associated device (0.0.xxxx). The other “views” of the hardware, e.g. sorted according to bus relationship or drivers, “reuses” this subchannel and device representation using symbolic links.

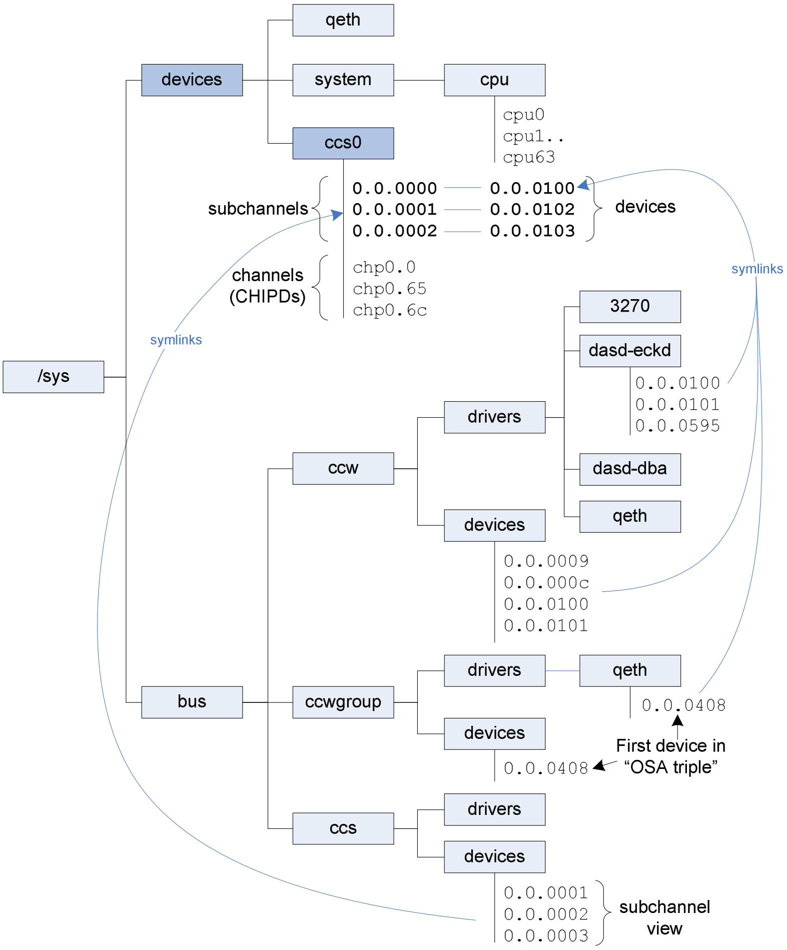


Figure 4-1: Excerpt from the “system file system” (sysfs) representation of mainframe hardware. The channels (chp0.XX), subchannels (0.0.nnnn), and devices (0.0.nnnn) shown, are itself directories containing “attributes” (files) to query/control the respective item.

The following shows the attributes related to an OSA (network connection) from the ccw**group** view. This “group” view includes “mechanism” like OSA network interfaces, which requires more than one device (subchannel) to function (see section 3.6.4 on OSA devices in z/VM). The three files cdev0-cdev2 are symbolic links to devices 0408-040A in "/sys/devices/css0/".

```
ls /sys/bus/ccwgroup/drivers/qeth/0.0.0408/
.                   cdev0            if_name             recover
..                  cdev1            ipa_takeover        route4
add_hhlen           cdev2            large_send          route6
blkt                checksumming     layer2              rxip
broadcast_mode      chpid            online              state
buffer_count        detach_state     portname            ungroup
canonical_macaddr   fake_broadcast   portno              vipa
card_type           fake_ll          priority_queueing
```

As an example of use, consider the following, which shows how the "checksumming" attribute can be used to determine whether TCP/IP checksum calculation is performed in the Linux TCP/IP stack (sw_checksumming) or by the OSA hardware. The final line shows how to enable hardware checksum calculation (assuming the necessary conditions are fulfilled):

```
# cd /sys/bus/ccwgroup/drivers/qeth/0.0.0408/
# cat checksumming
sw_checksumming

# echo hw_checksumming > checksumming
```

4.3 Drivers, S/390 tools and utilities

The mainframe specific drivers are distributed with the kernel source. They are typically located in the `/usr/src/linux/drivers/s390` directory. IBM, which is the maintainer of architecture specific source code, provides some rather detailed documentation for the "s390" specific drivers, which can be good to know: [23][40].

IBM also maintains a special package of user space tools and utilities, which provide easier access to the zSeries specific kernel features, sysfs entries and device drivers. The "s390-tools" package is normally included in the Linux distribution (in Red Hat as "s390utils") [59]. The following are some of the more "common used" tools provided with the package[2]:

dasdfmt: low-level formats eckd-dasds.

fdasd: creates or modifies partitions on (z/OS compatible) eckd-dasds.

dasdview: display DASDs and VTOCs information, or DASD content.

zipl: makes DASDs or tapes bootable.

tunedasd, adjusts tunable parameters on DASD devices.

vmcp, sends commands to CP (the Control Program) of z/VM.

[2] With SUSE Enterprise Linux Server 9 SP3, according to the documentation: /usr/share/doc/packages/s390-tools/README

4.4 Special considerations when running in VM

4.4.1.1 z/VM paging and Linux swap

When running Linux under z/VM, it can be useful to have a good understanding and an overall picture of the memory hierarchy for the combined environments. This can be useful in order to interpret performance measurements or peculiar system behaviour.

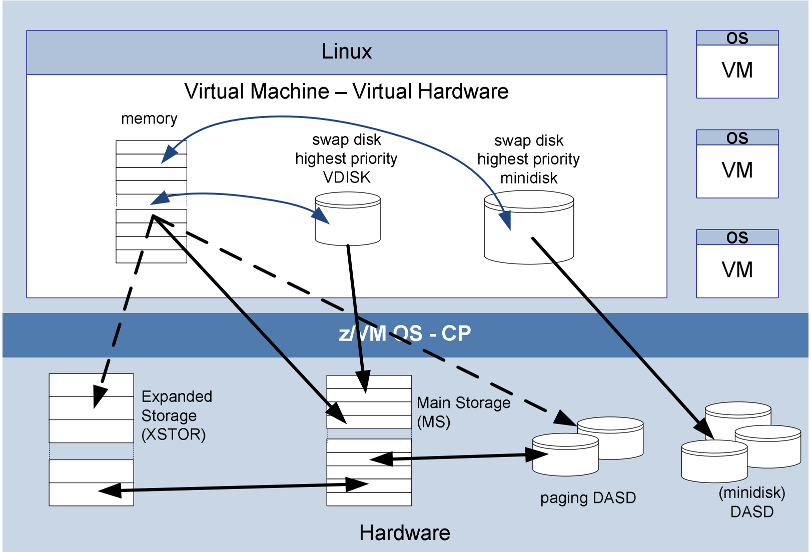


Figure 4-2: The many places Linux memory pages are scattered.

Linux itself uses **swap disks** to “enlarge memory”. In many cases Linux prefers to move rarely used pages (e.g. portions of loaded application programs or data) to the swap disks to facilitate space for file buffers and caches. That is why a smaller VDISK (virtual disk in storage) is often used as fast, high priority, swap devices under z/VM. (Actually to many VDISK might introduce a priority inversion problem, because CP prefers to steal pages from running guests rather than VDISKs, which are regarded as very important data areas for e.g. lock files. Special DCSS, **Discontinuous Storage Segments**, can be used as alternative swap device to avoid this problem. [59])

Now, CP also moves pages from main storage (real memory) to expanded storage or disk (paging DASD). The pages which Linux believes to be in memory might actually be located in expanded storage or on disk. Although the paging handled by CP happens transparently from Linux point of view, it might influence memory access or initialization time considerably.

Normally z/VM operates with **minidisk caches** in order to improve the performance of minidisk. Minidisks used for Linux swap normally has the minidisk cache disabled, because the lower priority swap disk otherwise also take up memory, or risk flushing more important pages from minidisk cache.

4.4.1.2 CPU usage readings

It is important to know, that 2.4 Linux kernels and older 2.6 kernels (e.g. the one in SLES9) uses "timer ticks" for internal time tracking and time measurement. These kernels are furthermore "unaware" that the processors might be virtual processors, and that the real processors below sometimes are dispatched to other virtual machines. This basically makes make CPU utilization readings from the `/proc` file system and programs like `TOP` untrustworthy.

To explain the problem, consider the situation in Figure 4-3. A single processor is alternating between kernel space and user space. The processor is active all the time, since this is NOT a virtual environment. The figure shows the exact number of time units spent in the two contexts; and the respective percentages of the total time.

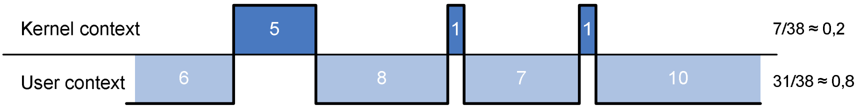


Figure 4-3: The exact number of time units spent in kernel and user space; the corresponding percentages of the total CPU time.

Now, the old kernels measure time in "time ticks". Depending on the exact timing conditions this can result in some degree of inaccuracy – see Figure 4-4. Although the time spent in kernel and user space is completely the same as before, the "sample rate" and the precise timing conditions influences the results considerately.

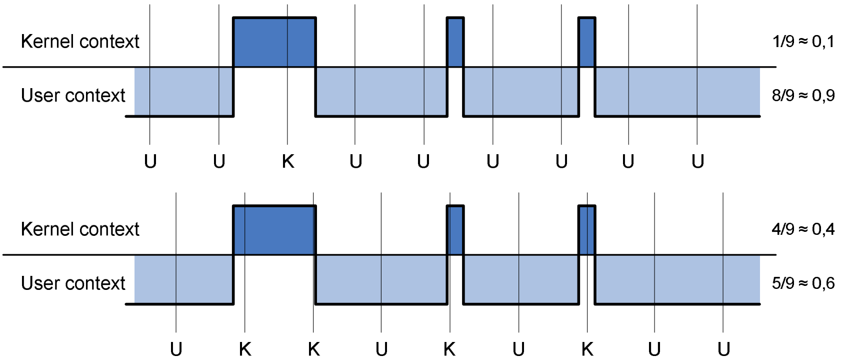


Figure 4-4: "Time tick" based accounting in two slightly difference timing conditions (same sample rate but difference "offset"). The distribution is exactly the same as in Figure 4-3, but tick based time measures differ considerately.

The problem becomes more complicated, when this type of time measurement is used in a virtual environment, where the hypervisor dispatches the processor to other virtual machines. Consider the situation depicted in Figure 4-5. The actual time spent in each context is now considerable less, since real processor is occupied elsewhere several times (white areas).

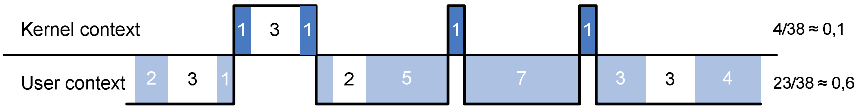


Figure 4-5: The exact time spent in kernel and user space is shown with colour. The white areas illustrate the time, where the real processor has serviced the hypervisor and/or other virtual machines.

Again consider the effect of tick based time measurements: The time ticks (interrupts), which occur while the real processor is dispatched elsewhere, are delayed and counted, when virtual processor regains control. The time "taken away" from the virtual processor is thereby not taken into account. Again depending on the exact timing conditions, the time measurement might be far from the truth: See Figure 4-6.

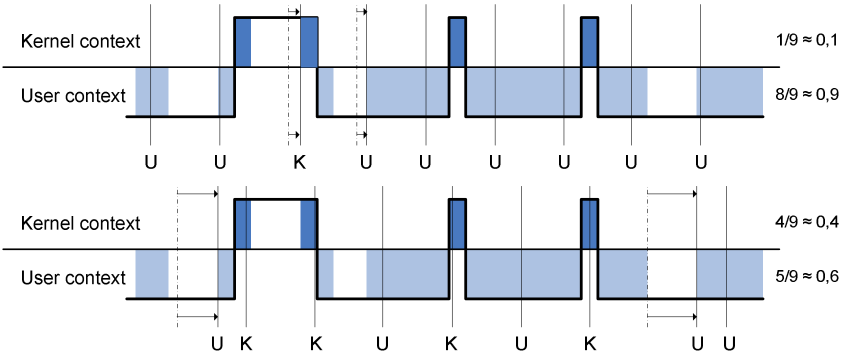


Figure 4-6: "Time tick" based accounting in a virtual machine. Two slightly difference timing conditions (same sample rate but difference "offset") yields different results and generally inaccurate results compared to the actual distribution in Figure 4-5.

In never Linux kernels (e.g. the one in SLES10) the problem has been resolved by introducing the concept of **steal time** and by eliminating time tick based accounting. The mainframe architecture offers a 64 bit timer register per *virtual* processor. It basically uses the same format as the "TOD (Time of Day) clock": the mainframe's Real Time Clock. But this virtual timer is only running as long as the virtual processor is running. The kernel can uses this register to keep exact track of the time, including the time spent in kernel space, in user space. But even more import it can keep track of the time, where the real processor has been dispatched elsewhere: Figure 4-7.

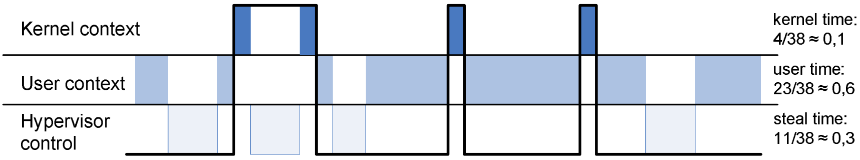


Figure 4-7: Timekeeping in never 2.6 Linux Kernels. A "real time clock" is used to keep extract track of time. The time taken away from a virtual processor is referred to as "steal time".

The new method has increased the precision of internal time keeping to at least 1 microsecond. In order to comply with old programs and interfaces, the interval time representation is converted to “ticks” (1/100 second), when the values are delivered to user space. The new concept of steal time can be read directly from the kernel via `stat` file in the `/proc` file system. Tools like `TOP` has been updated to include steal time as well.

It is very important to be aware of the current version of kernel and tools. This is not so much because of steal time itself, but the introduction of it also changes the meaning or semantic of the existing CPU readings: For example the user, system, nice, idle, and wait percentages shown by `TOP`.

In order to use the CPU reading from older system (like SLES9) it is necessary to provide supplement and trustworthy CPU reading from CP level. The Performance Toolkit (presented in section 5.1.2.3 on page 65) can provide such data. The toolkit can, amongst many other things, show the amount of processors resources allocated to each virtual machine, e.g. in terms of percentages of *real* processor time.

[51] [67]

5. PERFORMANCE TESTS & OPTIMIZATION

This chapter contains four performance tests or "system investigations", which first of all contributes to a better understanding of Linux, z/VM, and Linux under z/VM in particular. The chapter concurrently provides tangible test results, to generally accepted optimization suggestions and beliefs. The given recommendations sections are mainly company specific optimization suggestions.

The chapter is divided in four main sections each representing a performance test or system investigation. Test 1 to 3 mainly covers processors, storage (memory) usage, and disk I/O respectively. The fourth test basically combines the first three broadly examining the impact of adding a third virtualization level (running z/VM in z/VM in LPAR).

A number of monitoring tools, test programs and scripts have been developed in order to observe values of special interest and to be able to influence and stress the system in particular ways. The latter tests depend to some degree on tools, workloads, and programs (or derivations hereof) from previous test. The "method and tools" sections introduce these tools and programs consecutively; dependencies are explained and differences from the preceding tests are emphasized.

5.1 Test 1: Number of processors per Linux guest

5.1.1 Motivation

Several sources [5] [47, p. 30] indicate that the number of virtual processor per guest should be limited as much as possible, since unnecessary processors introduce an overhead. Some of the multiprocessor overhead is related to the "Diagnose x'44" instruction (or hypervisor call, see section 3.7.1 on page 47) [56].

The **Diagnose x'44'** or the "**Voluntary Time Slice End**" instruction is used from virtual machines with multiple virtual processors, when a spin lock exists. The scheduler is thereby informed that the reminder of the CPU time slice, which is allocated to the virtual processor issuing the instruction, is no longer useful. The priority of the virtual *processor* is thereby lowered compared to the other virtual processors within the same virtual machine. This will enable the virtual processor holding the lock to run and release the lock. [29]

Spin lock make no sense in setups with a single processor, since it does make sense to make the CPU actively wait in a loop in order to receive a lock, which only can be released by re-scheduling the processor to another process/thread. A Linux guest running in virtual machine with one CPU will therefore not call Diagnose x'44', but simple reschedule internally.

The hypothesis is that Linux guests, with mainly single threaded workloads, run slower in a multi processor virtual machine. It happen because Linux tends to use spin locks and thereby introduce Diagnose x'44' instructions, which takes longer time than internal scheduling in a single processor virtual machine.

This test is designed to find the optimum processor configuration for the Linux guests in a company specific z/VM environment. The test shows how the number of virtual processors influences the duration of real, typical and demanding jobs/workloads. The diag x'44 instruction counts is determined to verify that 2-CPU guests actually call the instruction.

5.1.2 Measurement methods and tools

5.1.2.1 QKUMON: Specialized monitoring in general

Linux provides many features and tools, which can be used when monitoring or testing the system. That is tools like `top` and `free`, not to mention the "raw data" provided by the kernel in the `/proc/` file system e.g. `meminfo`, `stat`, and `vmstat`. Unfortunately the output can be hard to interpret and comprehend in a "live" situation and the output format is seldom suited for subsequent data analysis. Naturally, dedicated monitoring products and programs exist to counter these difficulties. But in many cases such monitors products are completely overkill, their sample rate is to low or they miss customization capabilities to allow for highly specialized measurements.

Fortunately Linux systems (and UNIX variants) typically come with many small tools and capabilities, which make it possible to create a customized monitoring service with a relatively small amount of work. This particular approach has been adopted here.

qkumon

A monitor solution, named **qkumon** (by chance), has been developed for this project with the following in mind:

- Extreme customization abilities
- Near real time data delivery to allow real time graphing
- Data gathering option for later data analysis
- Simple data format allowing easy processing in standard tool (e.g. Excel)
- Connectivity to allow remote data gathering and graphing
- Simplicity is more important than avoiding communication overhead etc.

qkumon utilizes a mix of xinetd (the eXtended InterNET Daemon), pipes, scripts and values from `/proc` and programs like `top` as shown in Figure 5-1.

On the host being monitored `xinetd` has been configured to listen to a specific TCP port (here 5557) and to connect it with a script (`mon.sh`, see Appendix B.1.1). The `xinetd` service invokes the script when a tcp connection is successfully opened. Subsequently `xinet` redirect all data received from client to `stdin` and send all data written to `stdout` back to the client. The client can in other words control the script and receive the output from it. Since xinetd (or its predecessor `inetd`) is available in most Linux distribution this is an easy way to create a small highly customized server.

The actual monitoring process is instantiated and controlled by the client. It connects to the configured TCP port and `xinetd` invokes the `mon.sh` script, if the connection is successfully open. Now the client sends a character (or code) representing the wanted monitoring data, for instance a ‘t’ representing information from `top`. The `mon.sh` script parses input and invokes a corresponding command. This command performs the necessary measurements and reformats the output into a comma separated string (here using an `awk` script), which is written to `stdout` and thereby returned to the client.

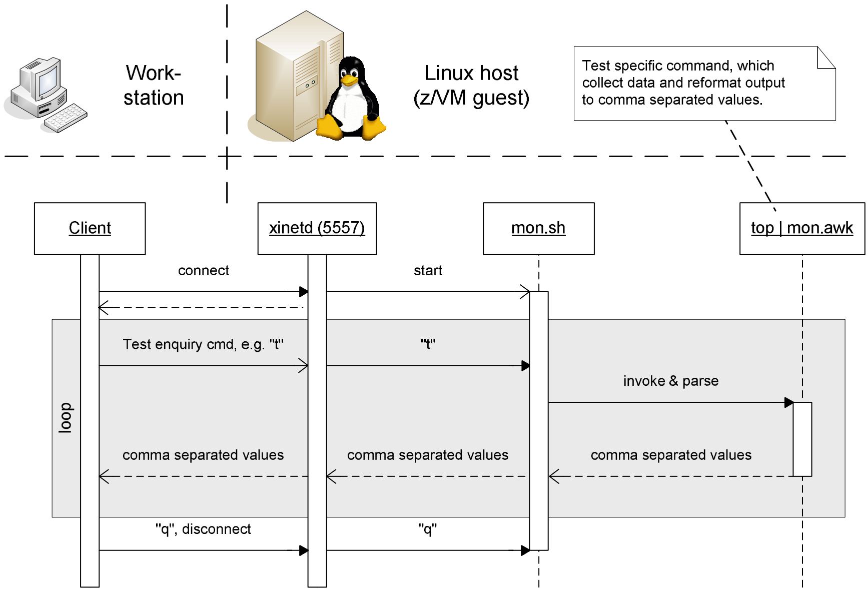
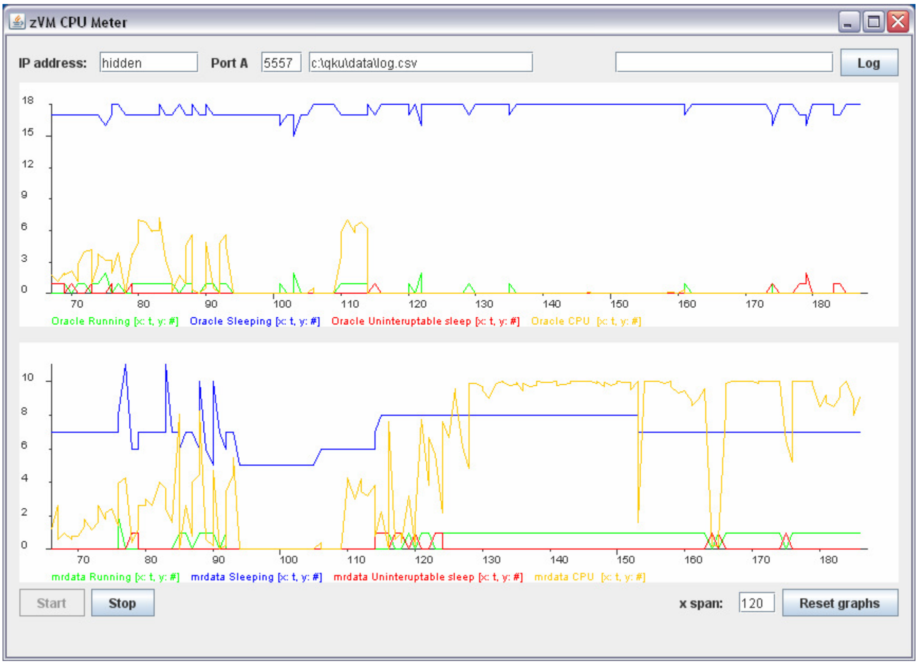


Figure 5-1: The monitoring setup constructed to deliver one of multiple sets of customized measurements to an arbitrary workstation via a TCP/IP socket.

Each time the client sends a new code, a corresponding data string is generated and returned. The client can easily parse the comma separated string and potentially graph the values in near real-time. Writing the unaltered data directly to a text file automatically generates a “CSV” file, which is easily readable for most spreadsheet programs for further analysis and later graphing. In this case, a simple client with graphing and logging capabilities have been developed in Java, but any type of program or script could be used. The GUI of the Java client program is presented in Figure 5-2 . (The source is not provided, but closely resembles the program for test 2 given in Appendix B.3)



zVM CPU Meter

IP address: hidden

Port A: 5557

c:\qku\data\log.csv

x span: 120

Log
18
15
12
9
6
3
0
70
80
90
100
110
120
130
140
150
160
170
180
Oracle Running [x: t, y: #]
Oracle Sleeping [x: t, y: #]
Oracle Uninteruptable sleep [x: t, y: #]
Oracle CPU [x: t, y: #]
10
8
6
4
2
0
mrdata Running [x: t, y: #]
mrdata Sleeping [x: t, y: #]
mrdata Uninteruptable sleep [x: t, y: #]
mrdata CPU [x: t, y: #]
Start
Stop
Reset graphs

Figure 5-2: The GUI of the developed Java client program, which is used together with the qkumon monitor. (Graphs are based on a class from the JChart2D library from http://jchart2d.sourceforge.net.)

When the client decides to stop the monitoring process it sends a stop code “q”. This stops the mon.sh script by breaking an internal loop. `Xinetd` automatically closes the connection to the client as the `mon.sh` script terminates.

5.1.2.2 Data gathering from within Linux

Test data for this particular test is mainly derived from the standard UNIX tool “`top`”. The top program provides a real time view of the running system, including both system summary information as well as information on individual processes. Especially the latter makes `top` useful in this particular case, since it provides easy access to the exact information needed.

The mon.sh-script invokes “`top`” in batch mode with one repetition, which disables the interactive user interface and sends a single output set to `stdout` (standard out). The output is piped into an `awk` script (`ora_mr.awk`, Appendix B.1.4.1), which parses it and generates a comma separated string including the following values:

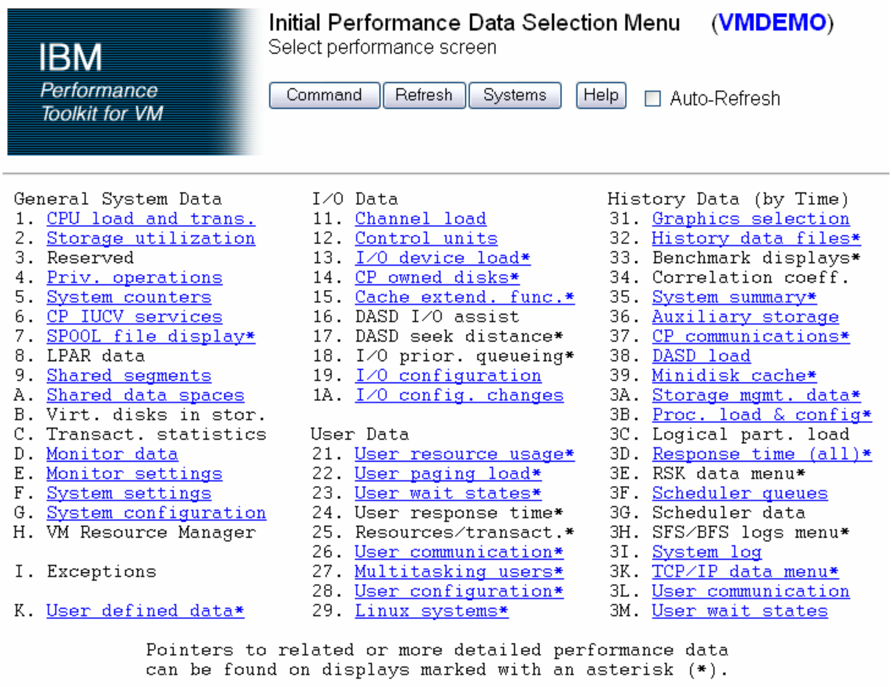
- Regarding processes owned by the application user (mrdata):
    - number of processes running, sleeping, in uninterruptible sleep
    - memory usage
    - % CPU usage (inaccurate “time tick based” value, see 4.4.1.2, p. 58)

- Regarding processes owned by the DBMS user (oracle):
  - number of processes running, sleeping, in uninterruptible sleep
  - memory usage
  - % CPU usage (inaccurate "time tick based" value)
- Number of other processes (not oracle or mrdata) running
- CPU usage related to other (not oracle or mrdata) processes

5.1.2.3 Data gathering outside Linux

The "IBM Performance Toolkit" is used to verify the CPU readings mentioned above, which basically cannot be trusted, since the present Linux version (SLES9 SP3) is unaware of the time VM spends servicing other guests (explained in section 4.4.1.2, p. 58).

IBM Performance Toolkit is a licensed product, which has to be purchased separately, although shipped as a component of z/VM. The Toolkit provides a variety of system information, performance measurements, and logging options (see Figure 5-3). Like all other things it runs in a virtual machine, but it depends on special data provided by CP, which enables it to present a picture of the complete virtual environment from "hypervisor view" (see Figure 5-4).



Initial Performance Data Selection Menu (VMDEMO)

Select performance screen

General System Data
1. CPU load and trans.
2. Storage utilization
3. Reserved
4. Priv. operations
5. System counters
6. CP IUCV services
7. SPOOL file display*
8. LPAR data
9. Shared segments
A. Shared data spaces
B. Virt. disks in stor.
C. Transact. statistics
D. Monitor data
E. Monitor settings
F. System settings
G. System configuration
H. VM Resource Manager

I. Exceptions

K. User defined data*

I/O Data
11. Channel load
12. Control units
13. I/O device load*
14. CP owned disks*
15. Cache extend. func.*
16. DASD I/O assist
17. DASD seek distance*
18. I/O prior. queueing*
19. I/O configuration
1A. I/O config. changes

User Data
21. User resource usage*
22. User paging load*
23. User wait states*
24. User response time*
25. Resources/transact.*
26. User communication*
27. Multitasking users*
28. User configuration*
29. Linux systems*

History Data (by Time)
31. Graphics selection
32. History data files*
33. Benchmark displays*
34. Correlation coeff.
35. System summary*
36. Auxiliary storage
37. CP communications*
38. DASD load
39. Minidisk cache*
3A. Storage mgmt. data*
3B. Proc. load & config*
3C. Logical part. load
3D. Response time (all)*
3E. RSK data menu*
3F. Scheduler queues
3G. Scheduler data
3H. SFS/BFS logs menu*
3I. System log
3K. TCP/IP data menu*
3L. User communication
3M. User wait states

Pointers to related or more detailed performance data
can be found on displays marked with an asterisk (*).

Figure 5-3: The selection menu from the "IBM Performance Toolkit" web interface.

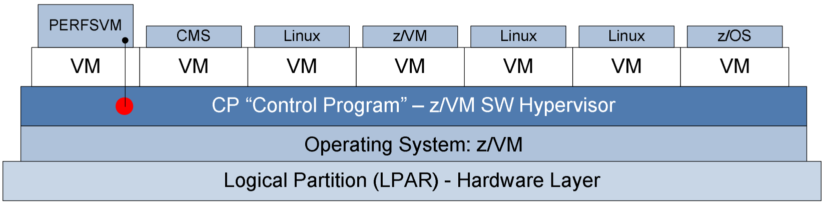

PERFSVM
CMS
Linux
z/VM
Linux
Linux
z/OS
VM
VM
VM
VM
VM
VM
VM
CP "Control Program" – z/VM SW Hypervisor
Operating System: z/VM
Logical Partition (LPAR) - Hardware Layer

Figure 5-4:The "IBM Performance Toolkit for VM" runs in its own virtual machine (typically "perfsvm") but the data is provided from CP.

The toolkit CPU readings are used to confirm, that the workloads actually are single threaded and that the virtual machines generally only occupy a single processor regardless of the number of virtual processors. The toolkit is also used here to generally verify that the tests are completed in a "stable" environment without big fluctuations caused by other guests.

Finally the Performance Toolkit is used to verify that the number of CPUs influences the number of diagnose x'44 instructions. Apparently is not possible to monitor this number per user. The "Privileged Operations" screen (#4) can however be used to manually supervise and record the current x'44 count.

Details about the test setup configuration can be found in Appendix A.1.

5.1.2.4 Monitors influence on the system

The specialized "qkumon" monitor does include periodic execution invocation of a number of processes (top, BASH, awk, xinetd) and it will influence the system to some degree. The influence is however considered to be insignificant since the polling frequency is very low (approximately 2 times per second) and the work is rather "simple" and sparse regarding CPU resources.

The influence of the Performance Toolkit is likewise considered to be insignificant. Especially since the toolkit runs as a VM guest itself it simply adds a little background noise in the big VM scheduling picture.

5.1.3 Workloads

The test is performed using workload definition WL1 and WL2, which have the following characteristics.

5.1.3.1 Workload 1 (WL1)

Workload 1 is "complete payroll processing" (payslip creation excluded): Several independent application executables are called during the payroll processing, which results in birth and termination of several new processes. Many of these programs individually access and "manipulate" the database.

The workload is of "mixed nature": A combination of CPU usage (application calculation + database queries) and disk usage (database, output file writing). The CPU usage typically fluctuates and rarely hits 100% (equivalent to one virtual CPU) for very long. For further details refer to Appendix A.1.2

5.1.3.2 Workload 2 (WL2)

Workload type 2 consists of Payslip PDF and PS file creation. It is dominated by a few sequentially running and very CPU intensive processes. The processing includes creation of PDF files from XML, and PDF to PS conversion. WL2 provides a more steady state / long run / high CPU load situation. For further details please refer to the workload definition in Appendix A.1.3.

5.1.4 Test description

The goal is to show, if the number of virtual processors influences the duration of the workloads above. To eliminate the influence from other z/VM guests (resource availability), the durations for the workloads are evaluated relatively: By running the same workloads in a 1-CPU guest and a 2-CPU guest concurrently. This also allows for the tests to be run in a production environment. The two guests running concurrently should be completely identical except for the number of CPUs. As a safety precaution the test is repeated with opposite CPU configuration on the same two guests in order to eliminate any unintended configuration differences.

As a supplement to these "relative" tests with concurrently running guests, the workloads are also run separately in one guest at the time: first in a 1-CPU guest and afterward in a 2-CPU guest. This is necessary to get the Diagnose X'44 instruction count, since is impossible to monitor this for individual guest. These separate tests can furthermore ensure that concurrent processing doesn't distort the results notably.

In order to be able to determine Diagnose x'44 overhead, the CPU `share` value for the 2-CPU guest has to have twice the CPU `share` for the guest with one virtual processor: Recall that z/VM divides a guest's relative CPU share between the guest's virtual processors (section 3.6.1.1 p. 35). If shares were not adjusted, it would be comparable to compare a PC with one 1000MHz processor and a PC with two 500MHz processors. Since the workloads in this test known in advance to be foremost single threaded, the 1000MHz PC would naturally outperform the 500MHz dual processor.

Figuratively speaking, this test should determine, whether the PC performs best with one or two 1000MHz CPUs – knowing that one of the processors will be mostly inactive and mainly be available for OS bookkeeping and background processing. In the virtual environment an extra processor results in an overhead (as mentioned), but the Linux system might still benefit from an extra processor for background processing. The test should show.

Table 5-1 gives an overview of different test settings.

Test	Linux Guest 1			Linux Guest 2				
number	CPUs	Rel. share	Workload	CPUs	Rel. share	Workload	Repetitions	Monitor
T1.1	2	200	WL1	1	100	WL1	4	qkumon
T1.2	2	200	WL2	1	100	WL2	4	qkumon
T1.3	2	200	WL1	-	-	-	2	qkumo, ptk: x'44 + %cpu
T1.4	-	-	-	1	100	WL1	2	qkumo, ptk: x'44 + %cpu
T1.5	2	200	WL2	-	-	-	2	qkumo, ptk: x'44 + %cpu
T1.6	-	-	-	1	100	WL2	2	qkumo, ptk: x'44 + %cpu
T1.7	1	100	WL1	2	200	WL1	4	qkumon
T1.8	1	100	WL2	2	200	WL2	4	qkumon
T1.9	1	100	WL1	-	-	-	2	qkumo, ptk: x'44 + %cpu
T1.10	-	-	-	2	200	WL1	2	qkumo, ptk: x'44 + %cpu
T1.11	1	100	WL2	-	-	-	2	qkumo, ptk: x'44 + %cpu
T1.12	-	-	-	2	200	WL2	2	qkumo, ptk: x'44 + %cpu

Table 5-1: Test 1; combinations of CPU and “relative share” settings together with workload name, the number of repetitions and monitor settings.

5.1.5 Test results and analysis

The CPU readings received from the Linux guests clearly confirm the mentioned workload characteristics. Figure 5-5 and Figure 5-6 give typical CPU readings for workload 1 and 2 respectively. Generally the readings for 2-CPU guests fluctuate more than the case for 1-CPU guest (compare mentioned figures to Appendix A Figure A1-2 and A1-3 on page A7).

It should be mentioned that the data illustrated have not been normalized using values from Performance Toolkit. It has, however, been confirmed, that CPU usage does not exceed 100% (~ 1 real CPU). An example of the Performance toolkit readings are given in Figure 5-6. Furthermore Performance Toolkit readings also confirm the general picture of a less intensive CPU usage for workload 1.

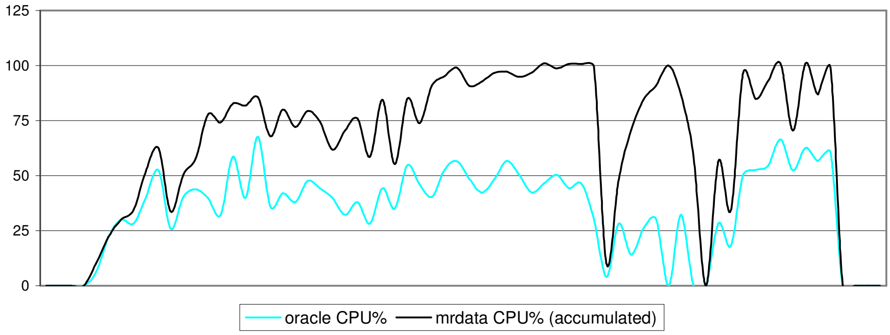


Figure 5-5: CPU usage characteristics for *WL1*. Contributions from oracle owned and application (mrdata) owned processes in accumulated (stacked) view. Performance toolkit readings confirm maximum CPU readings ~ 1 whole CPU. (Not normalized Linux readings, T1.1, 1-CPU-guest, workload during/execution time: 36s).

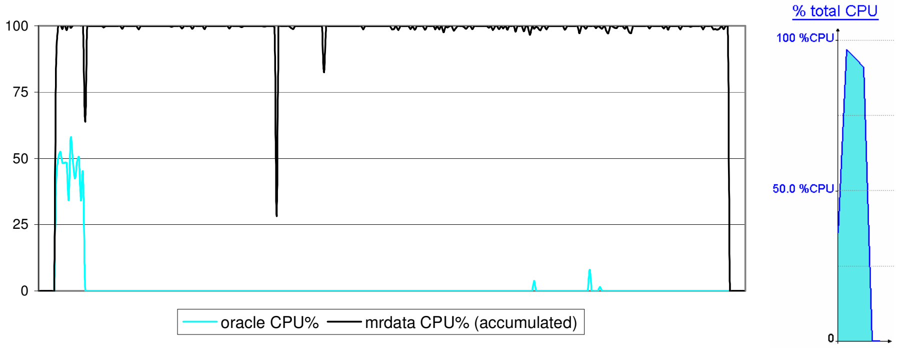


Figure 5-6: CPU usage characteristics for *WL2*. Contributions from oracle owned and application (mrdata) owned processes in accumulated (stacked) view. (Not normalized Linux readings, T1.2, 1-CPU-guest, workload during/execution time: 200s). Performance toolkit readings, shown in the graph to the right, confirm maximum CPU readings ~ 1 whole CPU.

All tests based on workload 1 show longer execution times on the guest running with 2 CPUs. The tests show execution time savings on the 1-CPU guest between 6% and 22% (see Table 5-2 and Figure 5-7).

T1.1	14%	**T1.2**	-4%	**T1.7**	6%	**T1.8**	1%
WL1	14%	**WL2**	2%	**WL1**	6%	**WL2**	0%
	22%		0%		6%		-1%
	17%		-1%		10%		1%

Table 5-2: Execution time savings in percent gained by using 1-CPU guests (in the test setups with two concurrently running guests). Generally no effect on workload 2.

Workload 2 is on the other hand not affected. Actually two of the tests does show slightly longer execution times on the 1-CPU guests (2 - 4%), but the general picture is that WL2 is unaffected by the number of CPUs (most values vary ± 1%).

Furthermore there is no reason to believe that the effect is related to the two guests running concurrently – influenced by the test condition that is. Although less trustworthy the measurement from the tests with one guest running give the exact same picture. One exception is test T1.5 and T1.6 (Appendix A.1.4.1). These tests, however, were carried out an hour apart and seem to be influenced by VM environmental matters.

The Diagnose x'44 readings (Appendix A.1.4.2) show, as expected, that guests with two CPUs issues many of these privileged instructions: The rate explodes with more than 5000 instructions per second for WL1 and nearly 3000 instructions per second for WL2.

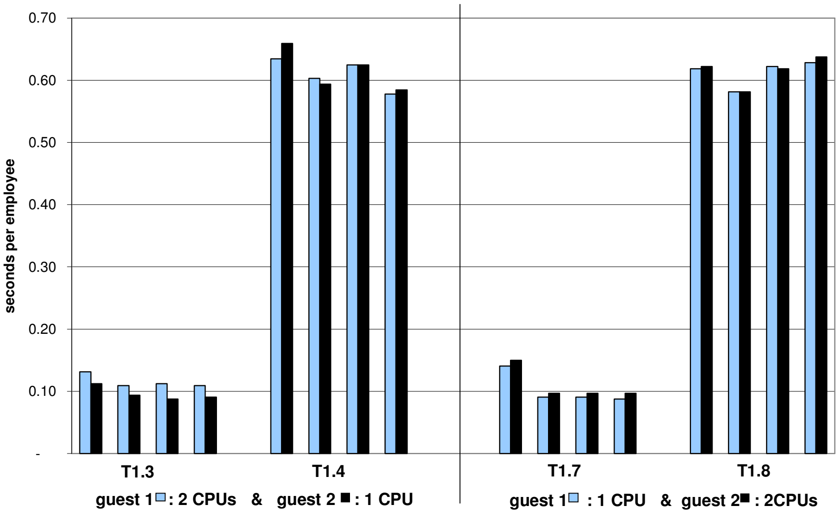


Figure 5-7: Workload duration measures. 1-CPU systems are generally faster on the database intensive workload which involves several sequential processes (T1.3 + T1.7). The difference disappears on the "pure" CPU intensive workload (T1.4 + T1.8).

5.1.6 Conclusion and recommendations

Given the test results above it has been shown that small improvements can be gained by reducing the number of virtual processors from two to one for Linux guests. The tests support the sources saying that the overhead is contributed by the Diagnose x'44 privileged instruction calls. This is evident since the execution time of WL2, which foremost run two longterm sequential processes and scarcely generates x'44's, is barely affected by number of CPUs. On the other hand WL1 is clearly affected and it generates numerous x'44's because of heavy process switching.

Given the actual workloads in the KMD environment it is recommended to remove the second virtual CPU from the Linux guests in production. Especially since workload 1 is considered much more typical than workload 2. If the workloads later on changes characteristics (to actual multiprocessing) the decision should be reconsidered.

Naturally a CPU reduction restricts the amount of work a multiprocessing guest can do. It does however also make the scheduling job easier for the hypervisor. It will always leave one CPU to achieve acceptable response time for others, while a guest generates pdf payslips, a Linux guest otherwise gets out of control or similar.

If both 1-CPU and 2-CPU guest are running in the same VM, the relative share for the 2-CPU-guests should be twice the relative share size for 1-CPU guest. That is, at least if 2-CPU guests are expected to finish single process workloads as fast as 1-CPU guests in CPU sparse situations.

5.2 Test 2: Memory usage

5.2.1 Motivation

The most common storage (memory) z/VM configuration guideline for Linux guests is to limit the main storage footprint for the guests as much as possible [59]. Linux is written to get the most out of the available system resources and therefore it eventually uses every bit of memory for buffers and file system cache as illustrated in Figure 5-8.

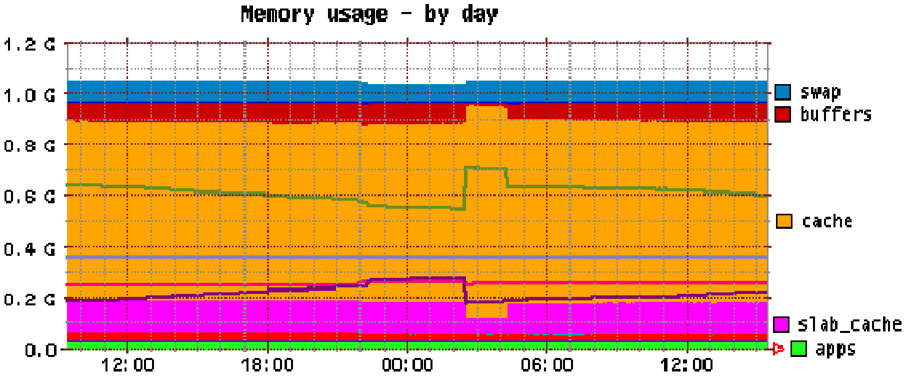


Figure 5-8: “Munin-graph” showing a typical memory distribution for a (inactive) Linux guest. Linux swaps although most memory is used for file cache and buffers.

In a virtual environment, where memory usually is overcommitted, there is not always a gain having large file system buffers in memory since the buffers risk being paged back to disk. Actually the effect might be a performance penalty since multiple guests suddenly are heavily dependent on the paging sub system and the same physical disks.

On the other hand a memory footprint, which is too small, can equally reduce performance significantly. When short of memory Linux “swaps” memory pages to disk to be able to complete its task. This naturally limit throughput drastically. The optimum solution is to adjust guest memory according the actual application needs.

This test or rather “investigation” is made to document and illustrate how Linux utilizes storage (memory) and how the z/VM environment is influenced accordingly. It examines pagning and “swapping” from both perspectives. The findings can be useful when sizing main storage for z/VM guests.

5.2.2 Measurement methods and tools

5.2.2.1 Data from within Linux

This investigation utilizes the developed monitor solution (qkumon) introduced in section 5.1.2.1. The actual data is derived from `/proc/meminfo` and `/proc/vmstat`. Generally the `/proc` file system contains files representing the current state of the kernel and the system. The `meminfo` and `vmstat` files give information on memory utilization and virtual memory respectively.

The data (plain text) from the two sources are parsed by two scripts (`memMeminfo.awk` and `memVmstat.awk`), which reformat the output to a comma separated string. The scripts are given in Appendix B.1.4.2 & B.1.4.3.

Memory model and /proc/meminfo data

Table 5-3 introduces the values derived from meminfo and how they are interpreted. Some confusion does seem to exist regarding some of these values especially swapCached. For instance an IBM Redbook [61] claims that SwapCached "reports the size of cache memory swapped out to swap devices". This surely does not make sense. The adopted understanding is mainly based on [3] and [9].

slab	Memory used by the kernel for caching different data structures.
PageTables	Memory used to map between virtual and physical memory addresses.
vmallocUsed	Kernel memory virtually contiguous but not necessarily "physically" contiguous.
Buffers	Relatively temporary storage for raw disk blocks.
Cached	Cache for files read from disk (the file cache).
MemFree	Unused memory generally available.
SwapCached	Memory that once was swapped out and has been swapped back in, but still resides in the swapfile.
SwapFree	Unused swap memory (on disk from Linux's perspective).

Table 5-3: Values extracted from meminfo by memMeminfo.awk and their meaning.

In order to ease graphic and subsequent data analysis, a few values are calculated and added to the output string in addition to values given directly in `/proc/meminfo`. The first value "apps" gives an estimate of the amount of memory occupied by running programs from Linux perspective. This value is calculated subtracting the other memory usage values from the total amount of memory:

$$\text{apps} = \text{MemTotal} - \text{MemFree} - \text{Buffers} - \text{Cached} - \text{SwapCached} - \text{Slab} - \text{PageTables} - \text{VmallocUsed}$$

Finally the value "SwapEvictedFromRealMem" is calculated to present the amount of pages, which are evicted from memory and only resides on swap disk. The value is calculated as the unoccupied swap space not taken up by swapCached, which in accordance with Table 5-3 still resides in memory. For simplicity the value is only denoted "swap" in following analysis section:

$$\text{swap} = \text{SwapEvictedFromRealMem} = \text{SwapTotal} - \text{SwapFree} - \text{SwapCached}$$

It should be noticed, that swapCached should be plotted twice (in memory and in swap), when plotting the memory distribution as a stacked "area chart" using the values above.

/proc/vmstat data

Table 5-4 show the four values derived from `/proc/vmstat` and their meaning. The focus is on page faults and page swapping in order to support the observations from `/proc/meminfo`. The values from `vmstat` are counters (in contrast to the values from `meminfo`). It is in other words necessary to have two consecutive values in order to plot these data in a meaningful way. In order to limit the complexity of the monitor script, which processes a single data set per invocation; such calculations are left to post processing.

pgfault	Number of minor page fault (since last boot).
pgmajfault	Number of major page fault (since last boot).
pswpin	Number of pages swapped in (since last boot).
pswpout	Number of pages swapped out (since last boot).

Table 5-4: Values extracted from /proc/vmstat by memVmstat.awk and their meaning.

5.2.2.2 Data from outside Linux

IBM Performance Toolkit is used to provide monitor data from z/VM. This is necessary since Linux is unaware of paging activity within CP. Remember that z/VM or CP typically over commits storage and is thereby forced to migrate some of the pages, which Linux believes is "real" physical memory, to disk (see section 4.4.1.1 on page 57).

The data of main interest is the paging behaviour of the VM guest under investigation. Toolkit screen 22 "User paging load" gives this information. The given details include paging activity (reads / writes), where pages resides (main storage, expanded storage, or DASD), and the rate of pages migrated from one storage type to another. The paging details for individual guest are normally not logged to file, but this can easily be accomplished by enabling user benchmarking (analogous to test 1, Appendix A.1.1.2). The monitor sample rate should in this case be set to 10 or 15 seconds to provide a more detailed view.

5.2.2.3 Monitors influence on the system

The monitors influence is considered to be neglectable. Deriving the data from /proc/ file system is fast and efficient. The parsing by awk and the processing by xinetd is rather simple and barely use any system resources; especially when the typical sample rate (approximate two sample pr. second) into account. The Performance Toolkit does not have significant influence on the investigation or system in general (see section 5.1.2.4).

5.2.2.4 Test tools: useMem

A small program has been developed to be able to influence and control memory utilization in an easy, reliable, and deterministic manner. The program is written in plain C using pthread as thread library. The main functionality simply is to allocate, initialize, and continuously use a variable amount of memory, while measuring the access rate. The source code is given in Appendix B.2.

Without start arguments the program starts by allocating and initializing 64MB memory and with 1 running worker thread looping through the allocated memory. The initialization (e.g. using memset) is quite important, since the kernel first allocates and occupies memory when used (not when allocated with "malloc").

When inputting "t" and "g" the program's main-function respectively increases and decreases the number of working threads. These working threads continuously access the allocated memory. The purpose with the working threads is as mentioned to keep the memory active. Each work thread simply loop though the allocated area one word at the time, reading data, performing a simple arithmetic operation (addition) and writing back the result.

If the memory is left unused Linux can safely swap the pages to disk. Using several threads can contribute to higher degree of "activeness". Furthermore the use of multiple threads makes it possible to evaluate how the number of threads influences the data access rate.

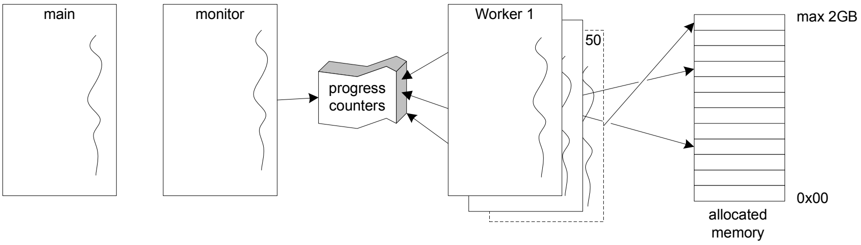


Figure 5-9: Central components of useMem test program. The main thread accepts input and increases/decreases memory usage and worker thread count accordingly. Worker threads continuously loop though allocated memory reading and writing new values. Monitor thread periodically outputs progress of worker threads.

The worker threads continuously increase their own individual counter variable, for every 128kb of memory they process. A monitor thread wakes up every second and calculates the progress of all the worker threads using these counter variables. The monitor thread calculates the throughput and sends it to stdout. It is presented as a comma separated string also including current time of day (hh:mm:ss), the program's elapsed time (millisecond accuracy), the current number of worker threads, and the current amount of allocated memory. All other messages are directed to stderr, which makes is easy to gather data for analysis and plotting.

5.2.3 Test description

This test requires a Linux guest, which has been running for some time and therefore has memory full of file caches and buffers. CP should furthermore have moved a substantial amount the guest's memory pages to DASD. The monitor tools mentioned above (qkumon and the Performance Toolkit) provide the necessary information to confirm these prerequisites.

The idea procedure for the actual test is simple: When the monitors are started, invoke useMem and allocate a good chunk of memory (around half of the guest's storage size). The guest becomes active: z/VM retrieves memory pages from expanded storage (XSTOR) and paging DASDs.

When useMem has allocated the memory, slowly increase the numbers of worker threads in order to determine the influence on throughput.

Stop useMem and restart it allocating a larger amount of memory, approximate 1.5 times the amount of memory allocated to the virtual machine. When finished, read or otherwise handle a large file to influence file buffers.

It should now be possible to find the effect on the memory initialization rate of already having pages in main storage. It should also be possible to see how Linux priorities data in different areas (cache, buffers, etc.).

To summarize (for a guest with 1G storage):

1) Check prerequisites:
   - Substantial amount of guest pages (> 50%) in XSTOR or on DASD
   - Linux should utilize most memory for cache and/or buffers.
2) Start monitoring:
   - Set Perf. Toolkit sample rate to 10 sec. and enable user benchmarking
   - Start memory usage monitoring (qkumon + client)
3) Invoke useMem program (uM) in guest and save output to file.
   Allocate 1/2 the guest's storage size: `./uM 512 >> testlog.csv`
4) Slowly increase number of threads to 5 (pressing 't' <Enter>)
   If possibly synchronize with Performance Toolkit measuring intervals.
5) Quit and reinvoke useMem program (uM) in guest and save output to file.
   Allocate 1,5 times the guest's storage size: `./uM 1536 >> testlog.csv`
6) Slowly increase number of threads to 5 (pressing 't' <Enter>)
   If possibly synchronize with Performance Toolkit measuring intervals.
7) Decrease number of threads to 1 ('g' <Enter),
   decrease memory to around the guest's storage size ('-' <Enter>)
8) Open large file ~512 MB (e.g. cat large file, >1GB, to /dev/null)
9) Stop useMem program (press 'q' <Enter>).

Stop monitoring (reset Perfkit sample rate to default, typically 60 sec).

The data from the Performance Toolkit, the qkumon monitor, and the output from the useMem program can now be correlated and analysed, for example with respect to:

- Memory initialization rate, correlated z/VM (CP) page migration and Linux page faults.
- How Linux priorities certain memory usages, e.g. file cache.
- How the number of threads influence the throughput.

5.2.4 Test results and analysis

Memory initialization

The three Figures on page 77 give three different views on memory handling, all covering the same period of time, namely step 1 to 5 in test description above or 5½ minutes. Figure 5-10 shows how Linux utilizes memory for different purposes. The figure confirms, that a large portion of memory is used for file cache before the test is started, - as required by the prerequisites.

At the mark "3)" the useMem program is activated and it starts initializing 512 MB of memory. It takes approximately 40 seconds to do the initialization, which corresponds to an initialization rate around 13MB per second.

Figure 5-11 shows the situation from z/VM (CP) paging perspective. As it could be expected, the paging system becomes very active in the same period. The reason is that Linux has used memory for file cache. Since the system has been inactive for some time, z/VM has moved the cache-memory-pages to DASD and expanded storage. Now the pages have to be paged back in by CP only for Linux to trash the content and make room for the useMem program.

Now compare the first initialization rate, at mark "3)", which the initialization rate for the second invocation of the useMem program at the mark "5)". Now the first 512MB is initialized almost instantaneously, or with an initialization rate in the order of 500 MB/sec. The clearly shows the speed difference and it explains why the storage size of guest generally should be minimized in order to reduce z/VM paging.

After "5)" Linux soon runs out of main memory and starts to "swap". The swapping activity is confirmed by "pswout/s" in Figure 5-12, which depictures the number pages swapped out pr. second.

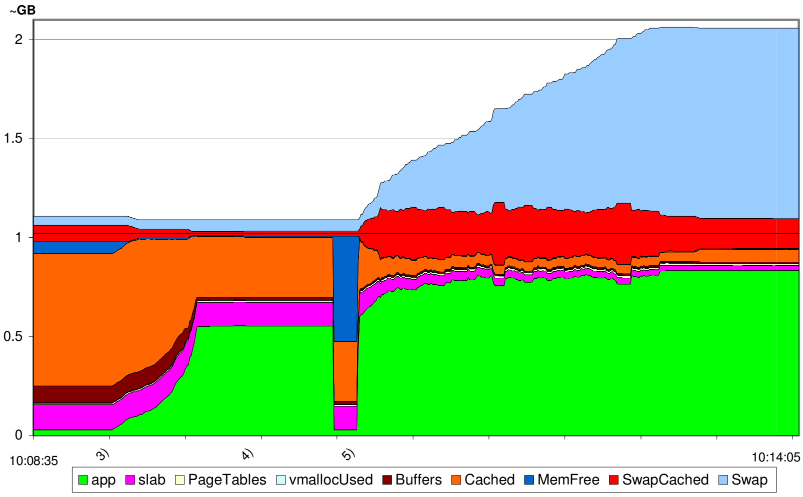


Figure 5-10: Memory allocation during the first 4 steps of the test procedure. Data derived from /proc/meminfo.

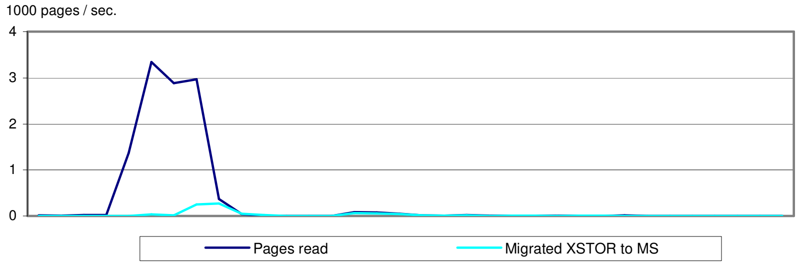


Figure 5-11: z/VM paging activity within same time interval as Figure 5-10. Data derived from CP (hypervisor) level using Performance Toolkit for VM.

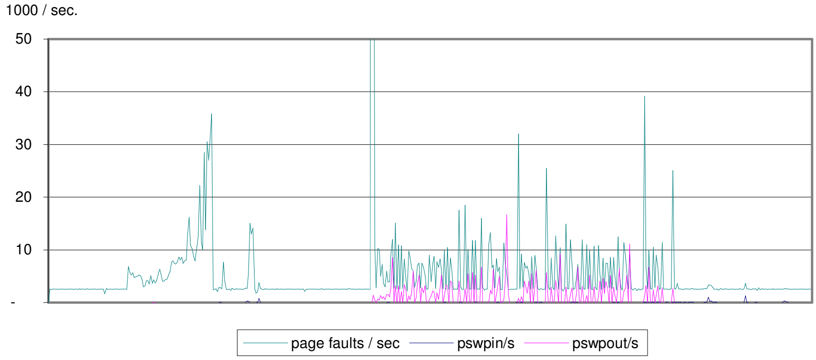


Figure 5-12: Linux swap activity within same time interval as Figure 5-10. Data derived from /proc/vmstat

Throughput

Figure 5-13 shows the benefit of having a real multiprocessor. The Linux guest used for the test had two virtual processors, which were supported by two real processors (IFLs). The "throughput" is doubled, when running with multiple worker threads.

It also shows that the throughput during test step 7 is much lower than during test step 4. This is obviously due to swapping, because Linux is forced to swap during step 7, where useMem uses more memory (1536MB) than the Linux guest has (1024 MB). This is supported by Figure 5-12 which show no swap activity during step 4; whereas Figure 5-15 shows a lot activity during step 7.

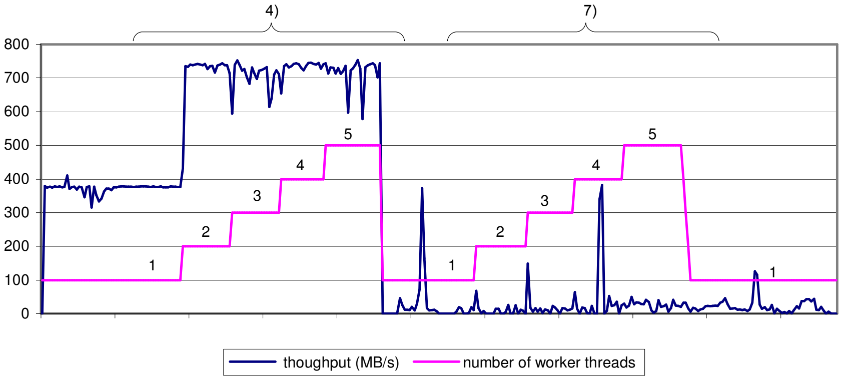


Figure 5-13: The throughput designates the amount of memory, which useMem is able to access and rewrite pr. second. Here depictured during test step 4 and 7 where the number of worker threads are increased to 5.

File cache

Notice on Figure 5-10 that Linux retain an amount of memory as file cache during step 5, although short of memory and heavily swapping. From Figure 5-14 it appears that the cache size is increased even further, during test step 8 where a large file is read.

Actually Linux' urge to swap in order to make room for file cache can be adjusting using the `/proc/sys/vm/swappiness` attribute. This issue will be revisited in test 3, where the matter becomes more evident.

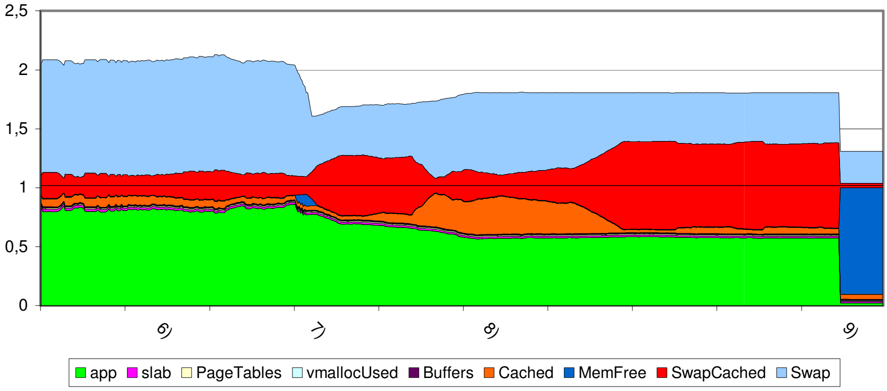


Figure 5-14: Memory allocation during step 6 to 9 of the test procedure. Data derived from /proc/meminfo.

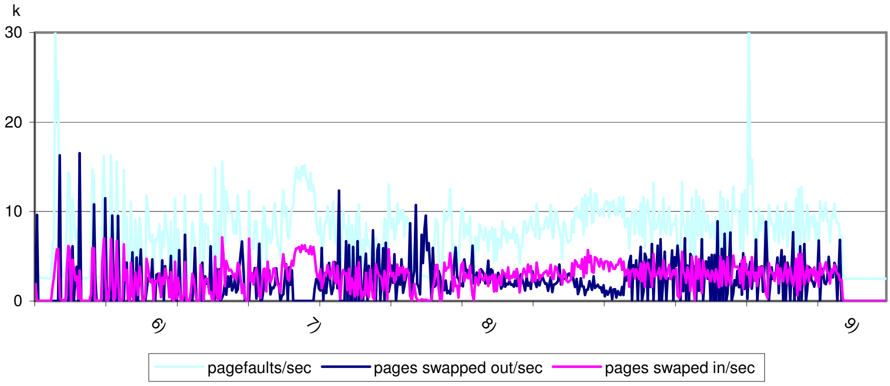


Figure 5-15: Linux swap activity within same time interval as Figure 5-14. Data derived from /proc/vmstat.

5.3 Test 3: Disk I/O; LVM stripes and caches

5.3.1 Motivation

The use of LVM (Logical Volume Manager) is very common method to improve disk performance on Linux systems. In very short terms, LVM can create virtual disk from several physical disks, and continuously spread data on all the physical disks in so-called stripes. In expense of a little book keeping overhead, this basically allows for the system to utilize several disks concurrently and thereby increase performance.

This test is amongst other things intended to show the effect of LVM on a mainframe system. Monitoring and optimizing of disk performance on a Linux z/VM guest can however be difficult task complicated by the many levels of buffers and caches: Linux has file buffers in memory (as shown in test 2), z/VM uses minidisk caches, and the disk control unit also contains large caches.

This test is therefore designed to show the effect of buffers, caches, *and* LVM stripes.

5.3.2 Measuring methods and tools

5.3.2.1 Test tools: Bonnie

This test relies on an old, open source, disk benchmark tool called bonnie, which very conveniently is included in the SLES9 distribution. Bonnie uses simple methods to measure the performance of UNIX file system operation. It basically creates a file of a specified size and uses different methods to access the file while measuring performance. An example of Bonnie output:

```
z6qku@linx03:/database/test03> bonnie -s 512
Mon Sep 24 15:18:29 CEST 2007
Bonnie: Warning: You have 996MB RAM, but you test with only 512MB datasize!
Bonnie:          This might yield unrealistically good results,
Bonnie:          for reading and seeking and writing.
Bonnie 1.4: File './Bonnie.10395', size: 536870912, volumes: 1
Writing with putc()...         done:  15224 kB/s  99.1 %CPU
Rewriting...                   done:  32375 kB/s  11.9 %CPU
Writing intelligently...       done:  28996 kB/s  18.7 %CPU
Reading with getc()...         done:  14815 kB/s  96.4 %CPU
Reading intelligently...       done: 470583 kB/s  88.9 %CPU
Seeker 1...Seeker 2...Seeker 3...start 'em...done...done...done...
              ---Sequential Output (nosync)--- ---Sequential Input-- --Rnd Seek-
              -Per Char- --Block--- -Rewrite-- -Per Char- --Block--- --04k (03)-
Machine    MB K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU K/sec %CPU   /sec %CPU
```

For further information please refer to http://www.textuality.com/bonnie/

5.3.2.2 Test tools: useMem

The useMem program from test 2 (see page 73) is reused here, in order to put the Linux file cache out of the game. This is basically accomplished by letting useMem occupy all available memory, and thereby leave no room for file cache. In order for this to work, useMem has to be started with a little higher priority than normal programs using nice:

```
nice -n -1 ./uM xxx 0      (as root)
```

It is also necessary to instruct Linux to minimize its tendency to swap. This is accomplished like this:

```
echo 0 > /proc/sys/vm/swappiness
```

5.3.2.3 Data gathering

This test relies on the performance measurement provided by Bonnie, which should be adequate to show the effect of buffers, caches, *and* LVM stripes. The qkumon memory monitor from test 2 (section 5.2.2.1 p. 71) is re-applied to reveal potential in memory file caching.

5.3.3 Test description

The idea is first to apply useMem, the swappiness attribute, and `nice` priorities as measures to avoid in memory file caching. The effect of this will be verified in the first three test steps (T3.1 – T3.3). Having file caching under control, Bonnie is used under the eight possible combination of:

- Enabled / disabled minidisk cache
- Enabled / disabled Control Unit Cache
- Running on LVM striped based disk / on non-LVM striped disk.

The combinations are given in Table 5-5. Bonnie should be run three times for every step in the test matrix.

Test number	useMem	Swappinesspiness	Minidisk cache	LVM stripes	CU cache
T3.1		60	x	x	x
T3.2	x	60	x	x	x
T3.3	x	0	x	x	x
T3.4	x	0	x		x
T3.5	x	0		x	x
T3.6	x	0		x	
T3.7	x	0	x	x	
T3.8	x	0	x		
T3.9	x	0			
T3.10	x	0			x

Table 5-5: Test matrix giving test combinations.

Swappiness is set by echoing the specified value to the attribute:

```
echo 0 > /proc/sys/vm/swappiness
echo 60 > /proc/sys/vm/swappiness
```

Minidisk cache can be turned off/on for a particular range of minidisk (0100-0110) in a specific virtual machine (LINX03), by issuing the following CP command from a privileged user/guest:

```
SET MDCACHE MDISK OFF USERID LINX03 0100-0110 DIR
SET MDCACHE MDISK ON USERID LINX03 0100-0110 DIR
```

Control Unit (CU) Cache can be controlled (-c) and inquired (-g) per linux disk device using tunedasd:

```
tunedasd -c bypass /dev/dasdf
Setting cache mode for device </dev/dasdf>...

tunedasd -c normal -n 2 /dev/dasdf
Setting cache mode for device </dev/dasdf>...

tunedasd -g /dev/dasdf
normal (2 cyl)
```

5.3.4 Test results and analysis

Figure 5-16, step 1) The system has plenty of free memory and can easily cache the complete file created by Bonnie.

Figure 5-16, step 2: The useMem program is activated and starts by consuming nearly all free memory. When Bonnie is started the system prefers to swap the useMem program to disk, even though it has been started with a slightly higher priority (nice -1). This is typical Linux behaviour and consequence of `/proc/sys/vm/swappiness` defaults to 60.

Figure 5-16, step 3: `/proc/sys/vm/swappiness` has now been set to 0. In combination with useMem having slightly higher priority and Bonnie slightly lower priority than normal, the effect is that the idle useMem now keeps its memory area. This leaves basically no room for Bonnie file cache. This is the setting used the remaining test steps (T3.3-T3.10).

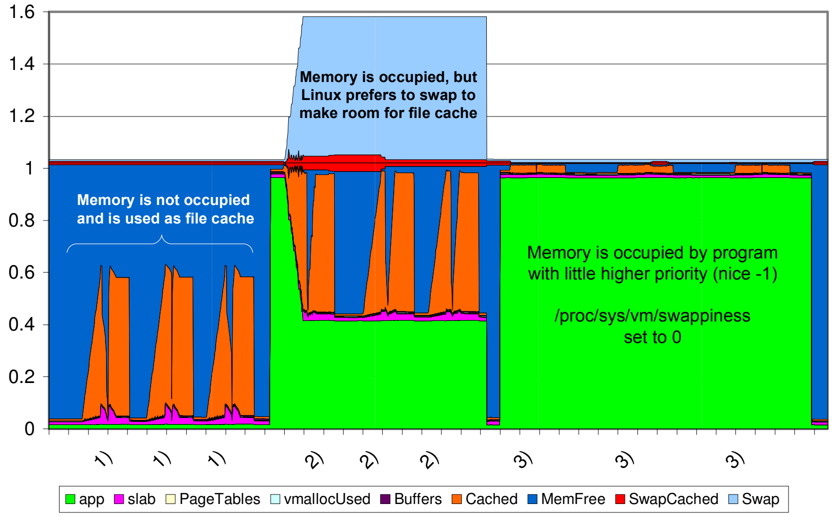


Figure 5-16: Memory distribution within the first three test steps (T3.1-T3.3).

The test results given Table 5-6 and Figure 5-17 generally confirm what could be expected: From T3.1 in one end revealing the immense advantage of files caches; to the T3.9 in the other end having no "support mechanism" and also the lowest transfer rate.

Test number	useMem	Swappinesspiness	Minidisk cache	LVM stripes	CU cache	Seq. write (block)	%CPU	Seq. read, rewrite	%CPU	Seq. read (block)	%CPU	Random read	%CPU
T3.1		60	x	x	x	137	47.5	110	25.9	1,041	99.6	61.7	100.3
T3.2	x	60	x	x	x	39	15.9	58	16.3	1,032	100.1	75.5	108.9
T3.3	x	0	x	x	x	34	19.8	32	12.8	431	82.0	21.9	136.0
T3.4	x	0	x		x	25	11.5	13	3.6	435	75.3	21.7	121.8
T3.5	x	0		x	x	44	21.3	25	8.8	126	21.6	2.6	10.4
T3.6	x	0		x		43	20.2	27	8.1	127	19.3	2.6	12.1
T3.7	x	0	x	x		44	21.0	27	9.2	115	18.9	2.6	10.5
T3.8	x	0	x			31	14.6	14	4.4	43	5.3	1.1	3.8
T3.9	x	0				31	14.5	14	4.4	43	5.1	1.1	3.2
T3.10	x	0			x	25	15.8	14	4.6	43	5.4	1.0	3.6

Table 5-6: Average values of the Bonnie test results (all test have been repeated three times). All read/write values are given in MB/second.

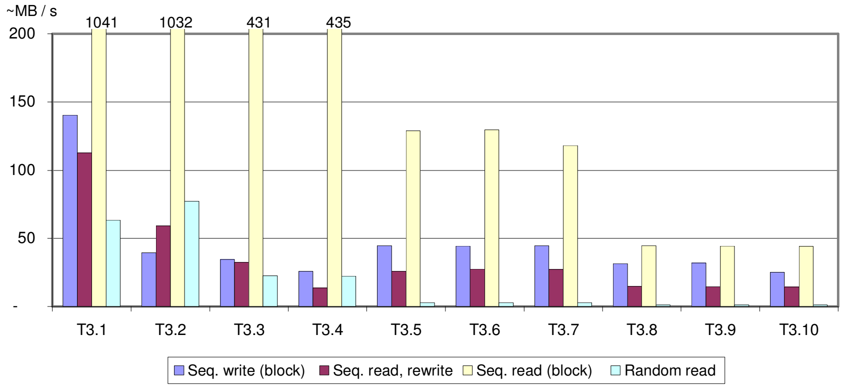


Figure 5-17: Plot of the transfer rates (read/write values) given in table above.

5.4 Test 4: z/VM in z/VM penalty

5.4.1 Motivation

The mainframe hardware supports two levels of virtualization (SIE) without notable performance degradation (as mentioned in 3.7.4 on page 51). Since the logical partitioning provided by PR/SM make up the first level, a z/VM system like KMD's production environment "KMDZVM", automatically uses the 2nd SIE level (refer to Figure 5-18). Any 2nd level z/VM system within the production system (like the VMDEMO system provided for the thesis project) will require a 3rd level of SIE, which has to be emulated.

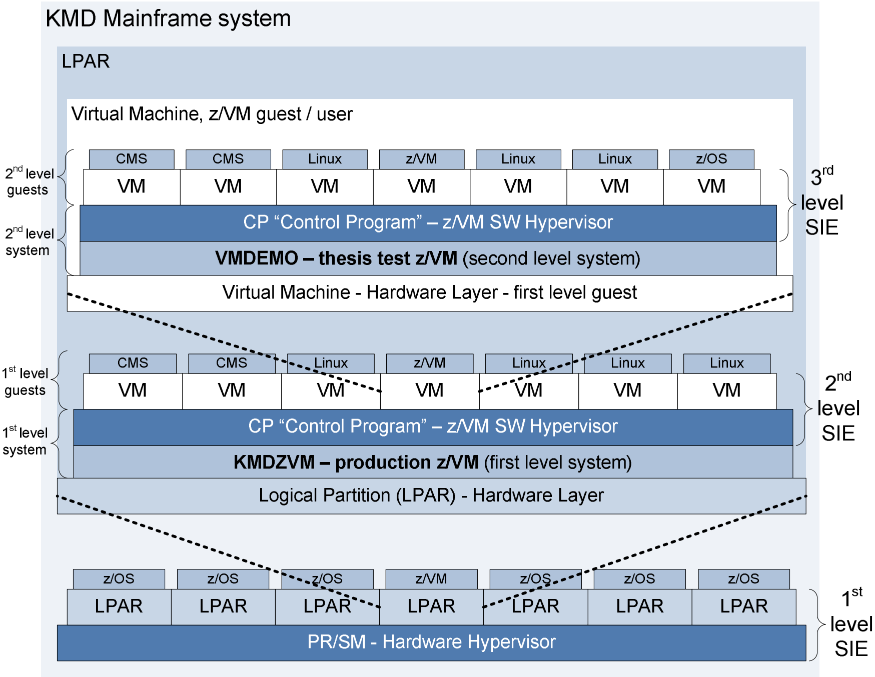

KMD Mainframe system
LPAR
Virtual Machine, z/VM guest / user
2nd level guests
2nd level system
CMS
CMS
Linux
z/VM
Linux
Linux
z/OS
VM
VM
VM
VM
VM
VM
VM
3rd level SIE
CP "Control Program" – z/VM SW Hypervisor
VMDEMO – thesis test z/VM (second level system)
Virtual Machine - Hardware Layer - first level guest
1st level guests
1st level system
CMS
CMS
Linux
z/VM
Linux
Linux
Linux
VM
VM
VM
VM
VM
VM
VM
2nd level SIE
CP "Control Program" – z/VM SW Hypervisor
KMDZVM – production z/VM (first level system)
Logical Partition (LPAR) - Hardware Layer
z/OS
z/OS
z/OS
z/VM
z/OS
z/OS
z/OS
LPAR
LPAR
LPAR
LPAR
LPAR
LPAR
LPAR
PR/SM - Hardware Hypervisor
1st level SIE

Figure 5-18: KMD's z/VM environment.

VM performance expert Bill Bitner states in [50] that running three levels of SIE is "fairly expensive". On the other hand he claims to have seen as much as 9 levels of virtualization, which imply that adding another layer shouldn't be fatal.

The use of the VMDEMO system for this particular project has indicated that emulated SIE indeed is very expensive. This test determines the effect of running a z/VM guest under z/VM in a LPAR. This test is created to reveal weather the overhead vary with the workload type and complexity.

5.4.2 Measuring methods and tools

5.4.2.1 Data gathering

The IBM Performance Toolkit for VM has been applied to provide processor utilization data for the entire z/VM system – or actually from both z/VM system (KMDZVM and VMDEMO). This approach has been possible, because the test was completed during a weekend in completely idle production environment, where any processor usage above 5% could be attributed to the test.

5.4.2.2 Test tools

The tests are based on the same workloads and tools, which have been used in the preceding tests:

- From test 1: The database intensive payroll processing workload 1.
- From test 2 and 3: The useMem program applied as a multi threaded processor consuming, but yet simplistic workload (no complex instructions involved is involved) .
- From test 2: “CAT’ing” a large file to /dev/null in order to produce simple file I/O.
- From test 3: The Bonnie benchmark program in order to produce a more complicate file I/O.

5.4.3 Test description

The four test steps given in Table 5-7 have to be completed in similar Linux guests within the two z/VM systems (or preferably in the same guest moved from one z/VM system to the other).

The performance monitor should be active in both z/VM system, when monitoring the second level *system (VMDEMO)*.

T4.1	Test 1 WL1: Perspektiv Payroll Processing (LS410)
T4.2	`useMem/uM 512 2` (running in 3 minutes)
T4.3	`cat /database/pas.dbf > /dev/null` (~ 2GB)
T4.4	`bonnie -s 1024`

Table 5-7: Test steps to be performed in both z/VM systems.

5.4.4 Test results and analysis

The test results (Figure 5-19 to Figure 5-22) reveal an immense overhead in the 2nd level z/VM system (VMDEMO). As shown in the first figure, it only takes KMDZVM around 9 minutes to process all tasks. As illustrated in the next figure, it takes approximately 36 minutes to do exact the same in VMDEMO.

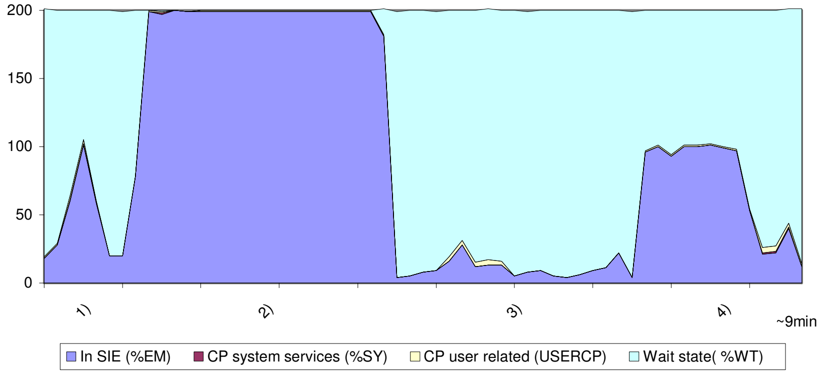


Figure 5-19: Test T4.1 to T4.4 run in first level z/VM system (KMDZVM); measurements from the CP level (collected by Performance Monitor) *in the same system*.

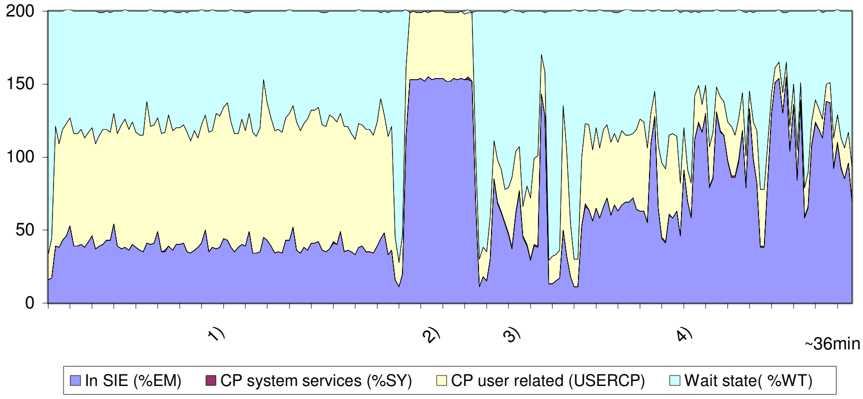


Figure 5-20: Test T4.1 to T4.4 run in second level z/VM system (VMDEMO); measurements from the CP level (collected by Performance Monitor) *in KMDZVM*.

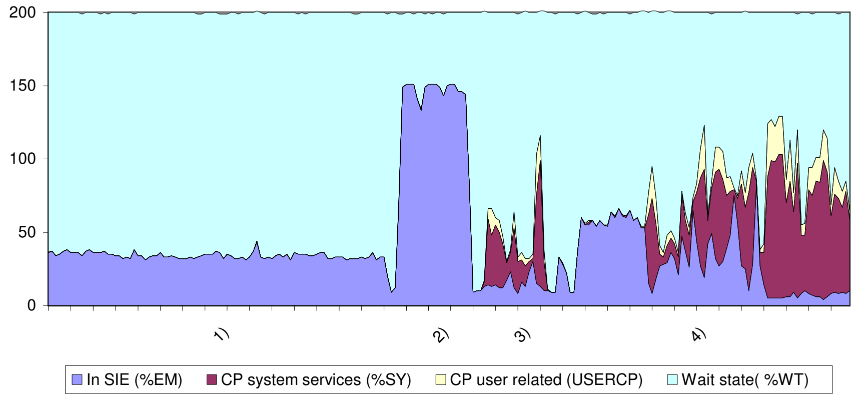


Figure 5-21: Test T4.1 to T4.4 run in second level z/VM system (VMDEMO); measurements from the CP level (collected by Performance Monitor) *in VMDEMO*.

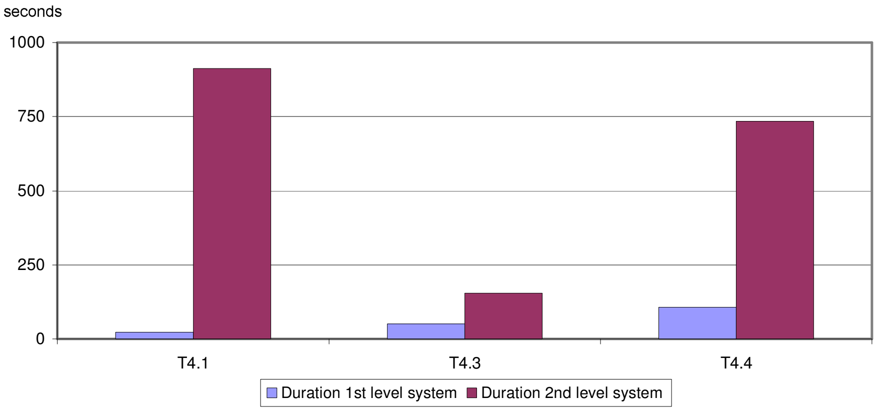


Figure 5-22: Comparison of runtimes/the duration of the individual test steps in the two z/VM systems.

Figure 5-22 gives a clear impression of, that different workload types contributes to different degrees of overhead. It is the database intensive (complex) workload in test step T4.1, which is affected the most. This is supported by Figure 5-20 which shows (in yellow), that T4.1 is the workload, which contributes with most work for the hypervisor (CP) in the first level system.

5.4.5 Conclusion and recommendations

Based on test results it can be concluded, that a 2nd level VM system *never* should be used for production, - simply being to ineffective. Furthermore a 2nd level test system should always be limited on processor resources, since otherwise innocent background processes suddenly can be turned into resource consuming problems.

6. AVAILABILITY - MONITORING

This chapter deals with the issue of availability: In this context it is the matter of keeping software services available for the customers. In accordance with KMD's wishes, this is considered from a monitoring angle: The ability to automatically detect issues, which hinders or possibly will hinder the customer from using the system with satisfactory performance.

Monitoring of hardware and network devices plus software services, are important tasks when managing an IT infrastructure. Companies and larges enterprises, which are greatly dependent on IT or maybe even profit on providing them, often have a dedicated central monitoring centre. Here people supervise the operation of the complete IT environment 24x7. In this way it possible to react very quickly when an issue arises – and hopefully resolve them before they become real problems.

Regardless of the size and complexity of the system(s) being monitored, it requires some kind of programs or systems to do the job. This chapter reflects on different aspects of finding the right monitoring software for Linux guests under z/VM on the mainframe (in some aspects particular for KMD).

6.1 Monitoring methods

Monitoring systems fall into two groups depending on method used to gather data. The first group relies on a **centralised approach**. Here a single program or a few processes in a single location collects data. The central monitoring system actively tries to determine the status of the monitored entities. This is accomplished simply by establishing some kind of connection to them or by performing operations involving them (Figure 6-1).

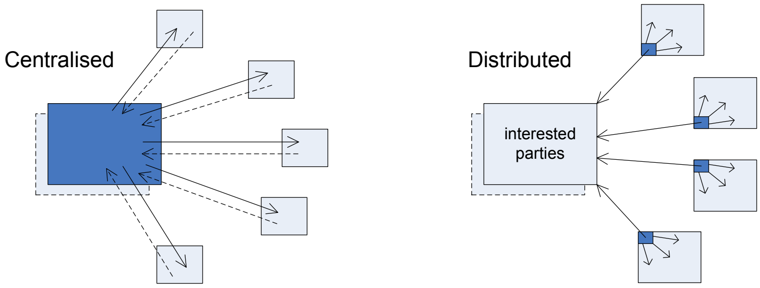


Figure 6-1: Centralised versus distributed monitor approach.

The second group is based on the **distributed approach**. Here special monitoring programs are placed on or close to the monitored entities. These programs are often called "agents" and they perform the necessary tests and report the outcome to the relevant entities. These agents can range from simple programs to complex intelligent autonomous mobile software elements."

[68]

6.2 Components to monitor

6.2.1 Standard warning / error conditions

This section presents typical conditions, which result in warning or errors on normal (not virtualized) servers, e.g. Microsoft Windows servers, and UNIX servers. It does not take duration, number of repetition, the severity of event into account, although such thresholds/levels typically are specified.

Processor

Processor usage is often monitored in order to detect system irregularities, or general resource shortages. Concrete warning condition might include:

- High system (kernel) CPU usage, e.g. above 80%
- Generally low idle time (might indicate CPU overload/resource shortage)

Memory

Memory is naturally also an important resource. It is a severe problem, if a system runs out of memory. It can have a great negative impact on performance if a system constantly needs to swap, because the active working set is bigger than the physical memory. Examples of error/alert conditions:

- The system is low on free swap space and thereby on memory.
- The page rate to/from swap is too high.

Disk, file system, and files

Disks are monitored in order to ensure space for persistent as well as temporary data. In some situation monitoring might include the physical well-being of the disk devices, in order to ensure data integrity. Unexpected periods with high amounts of reads/writes might indicate an I/O bottleneck or a failing application out of control. Individual files might also be monitored for changes to reveal security breaches or because the files are crucial for system operations. Concrete examples of warning/alarms in this category:

- Disk low on available space, e.g. below a threshold in MB or a percentage.
- Disk low on i-nodes, e.g. below a certain percentage of available i-nodes.
- High disk utilization, “too many” reads or write.
- Any modification of files, e.g. `/etc/passwd` and `/etc/group`.

Network interfaces and connectivity

A generally well-functioning server is not worth much, if the services running on it cannot be reached, because the network is malfunctioning. Examples, which could raise warning or errors:

- High Packet loss.
- Many packet checksum errors or packet collisions
- Network interface down.

Programs, processes, services and logs

The last category focuses on individual programs. If a key program, process, or service is not found on the process list as expected, this is a clear indication of an error. In other situations it can be useful to follow (filter) log files to reveal error conditions. Some times it can even be necessary to invoke the actual service and ensure it gives the expected results with an adequate response time.

- Unexpected high processor utilization of individual programs/daemons
- High number of zombie processes
- Special entries appear in log files
- A service does not respond as expected within acceptable time.

6.2.2 Mainframe/virtualization considerations

Running servers in a virtual environment like z/VM require some special consideration of what to monitor, how to do it, and from where to do it.

Processor

Processor usage readings are a good example of monitoring data, which cannot be delivered from within the Linux guest. Although never Linux kernels include “steal time” (as mentioned in section 4.4.1.2 p. 58) this is definitely not the case with older kernels. In virtual environments still dependent on older kernels, is therefore necessary to trustworthy CPU usage readings from the CP layer.

Hypervisor data

Monitoring from within individual guests (Linux or not), will generally not be able to reveal much useful information about the virtual environment (the z/VM system) as a whole. Only data from the hypervisor level (from CP) can provide a trustworthy picture of all the virtual machines together. Important values include CP paging activity, I/O rates, and the overall CPU utilization.

Luckily z/VM provides a general method for accessing CP monitor data for programs like the “IBM Performance Toolkit” and other monitoring products.

Instrumentation overhead

When running virtual servers, focus on resource sharing and consumption typically increases, and it becomes much more apparent, if the monitoring software squanders processor cycles and memory away. If the instrumentation overhead for a single Linux instance come close to a feeble 1% of a processor, the monitoring system alone will quickly consume half or whole processors depending on the number of guests.

Adding extra processors will increase the license costs of the remaining software portfolio, and the monitoring software suddenly becomes incredible expensive. Monitoring solutions should therefore be small, fast and efficient and preferably have a small memory footprint.

6.3 Other requirements

6.3.1 Existing Alert Chain

The monitoring product has to be able to integrate into KMD's existing monitoring system. KMD operates many different hardware platforms and operating system and correspondingly several monitoring solutions. Additional monitoring products have been added as KMD has taken over operation of already existing IT systems using non KMD standard software.

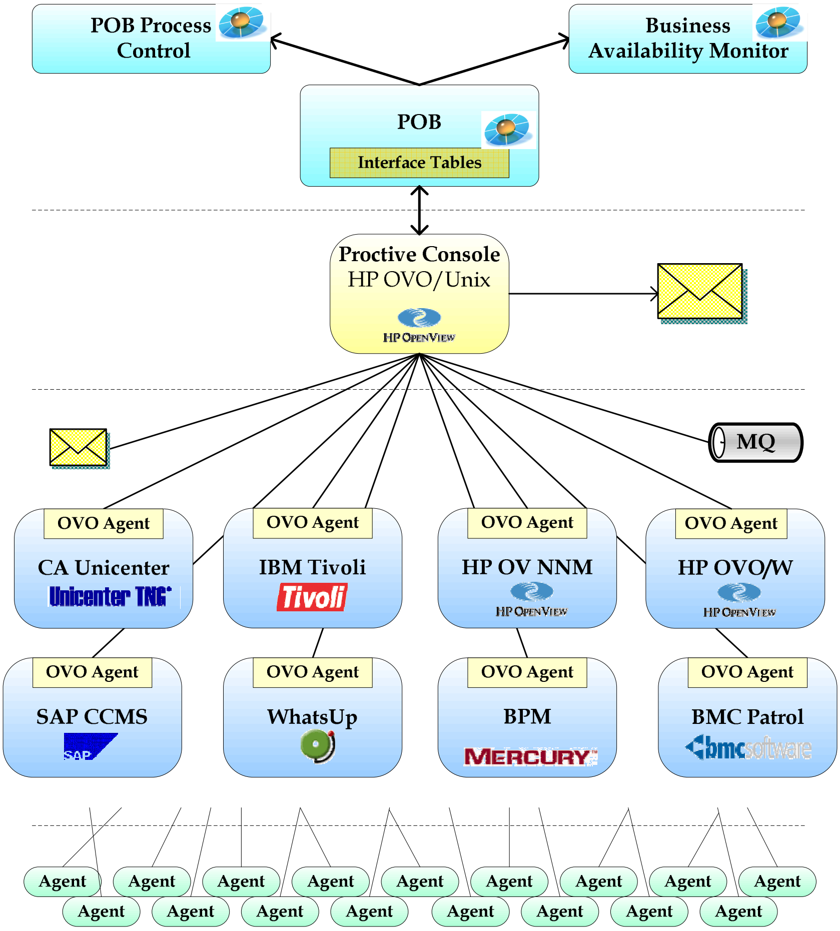


Figure 6-2: The "alert chain" at KMD. Monitoring messages (alerts and warnings) are transformed to a uniform format and gathered in HP OvenView/Unix.

In order to provide the central monitoring centre with single view of all platforms, KMD has chosen to gather all alerts in HP OpenView. The individual monitoring solutions have a central server gathering data for their specific environment (see Figure 6-2). These central servers transform alerts into a uniform message format developed by KMD. HP OvenView agents on the central servers then transfer the messages to the main HP OpenView monitor (the Proactive Console) using in build-in transfer mechanism.

Besides providing a single view of all alters, HP OpenView furthermore has the ability to “forward” alerts to POB (Wendia Point Of Business). POB is software tool, which amongst other things is used manage the IT Service Management processes within the company. (KMD is organized according to ITIL, the IT Infrastructure Library, which is a best practices framework provides guidance on how to manage IT infrastructure and to streamline IT service[2].). The monitoring system is in other words able to automatically invoke the business process, which can rectify the problems.

6.3.2 Support, manageability, security, costs

There are naturally other issues, which should be taken into account when choosing monitoring software: Product **support** for once. It is imperative for a company like KMD that there is a trustworthy and reliable a support unit, which can answer question, help solving problems reasonable fast, and provide fixes to potential software bugs. This can be a problem in regards to open source software, unless the software is included in one of the enterprise class Linux distributions and supported there.

As the number of monitored system raises is becomes apparent, that **easy management** is another important factor. The task of logging in on potentially hundreds of servers, in order to update the monitoring software or make small configuration adjustments, quickly becomes time-consuming and fatally monotonous. It can therefore be an important feature, if the monitoring system includes some kind of central management, if not already included in the system being monitored.

Monitoring software can also pose a potential **security** risk. It can for example be necessary to loosen firewall rules in order to allow transfer of monitoring data and/or managing the monitoring software itself. High security should therefore an integral part of the monitoring solution.

Finally the **costs** should be evaluated – and not from a purchase and software licence perspective alone. Implementation, customization, and management costs should be assessed as well.

6.4 Monitor Software Candidates

This section gives an initial list of monitoring software candidates, which deserve further investigation. The candidates are ranging from relative simple open source projects with Linux-only monitor scope, to complex software monitoring products, which monitors both z/VM and Linux.

6.4.1 Commercial software

IBM Tivoli OMEGAMON XE

Tivoli OMEGAMON XE is IBM's series of performance and availability products for the mainframe platform. The products were originally produced by the Cradle Coorporation, which was acquired by IBM. The series include OMEGAMON XE products for "z/VM" and "Linux for zSeries".

The z/VM monitor software provides amongst other things a quick view of z/VM system health; workloads for virtual machines, response times, throughput, and operational errors.

OMEGAMON XE for Linux on zSeries provides detailed performance metrics, such as CPU use, I/O statistics, network performance, and other from important systems, which help administrators see crucial interdependencies.

IBM Tivoli OMEGAMON XE for Linux on zSeries integrates with Tivoli OMEGAMON for z/VM to provide the crucial combined dimension.

http://www.ibm.com/software/tivoli/products/omegamon-xe-linux-zseries/

http://www-306.ibm.com/software/tivoli/products/omegamon-zvm/

[10]

6.4.1.1 Velocity software ESALPS

ESALPS is Velocity Software's "Linux Performance Suite" for Linux, TCP/IP, and z/VM. The products included in the suite can e.g. collect and display network, Linux and z/VM performance data; create reports and perform detailed analysis.

ESALPS provides exception monitoring and reporting, which should allow for fixing problems before noticed by the users. The ESALPS whitepaper directly state, that ESALPS can provide operational alerts, which can be displayed on an HP Openview console.

http://www.velocity-software.com/esalps.html

[32][33]

6.4.1.2 BMC MAINVIEW for Linux – Servers

"MAINVIEW for Linux – Servers" is part of BMC's MAINVIEW family, which includes products for zSeries system management and intelligent optimization of the mainframe infrastructure.

The product should be capable of monitoring hundreds or even thousands of Linux systems in a z/VM environment. The key Linux areas monitored include: system activity; process activity; user information; shared memory, message queue, and semaphore statistics; file system information such as space utilization, i-nodes, and block information. The product should also be able to monitor z/VM hypervisor performance.

http://www.bmc.com/products/proddocview/0,2832,19052_19429_26309_8716,00.html

[70]

6.4.2 Open Source Linux Monitoring software

6.4.2.1 Nagios

Nagios is a host and service monitor designed to detect problems before they are noticed by the end-users. It is designed for Linux but works fine under most UNIX variants. A monitoring daemon runs intermittent checks on hosts and services specified by external "plugins”. The daemon can send notifications (email, instant message, SMS, etc.) when problems are encountered. Nagios allows for accessing status information, historical logs, and reports via a web browser

Nagios is included in the Novell SUSE Enterprise Linux Server 9 and 10 distributions in version 1.2 and 1.3 respectively. Novell offers Level 3 support (code debugging and patch provision) on the software [43].

http://www.nagios.org/

[42]

6.4.2.2 MON, Service Monitoring Deamon

“mon” is a service availability monitoring tool, which can send “alerts” on prescribed events. The “mon” distribution comes with a predefined “monitors” ranging from simple ICMP echo (ping) to IMAP and LPD Print Server supervision. A monitor can also include a complex analyzing of results from a application-level transaction. “Alerts” are actions such as sending emails, making submissions to ticketing systems, or sending events to HP Openview management stations.

The “mon” tool is included in the Novell SUSE Enterprise Linux Server 9 and 10 distributions. Novell offers Level 2 support (reproduction of potential issues) on the software [43].

http://www.kernel.org/software/mon/

[41]

6.4.2.3 Hobbit

The Hobbit monitor is intended for monitoring of servers, applications and networks. It provides real-time monitoring, an easy web-interface, historical data, availability reports and performance graphs. To ease configuration, Hobbit keeps all configuration data in one place: On the Hobbit server. It is in other words not necessary to log in on all servers to make changes to the configuration.

Hobbit is especially interesting in this context because a z/VM Hobbit client is available. Hobbit is unfortunately (being open source) not included in the Novel SUSE Enterprise Server distributions.

http://www.sourceforge.net/projects/hobbitmon

[72]

7. CONCLUSION

Having started from scratch, with a typical limited academic understanding of mainframe architecture, this thesis project has provided a basic insight in, and appreciation of, IBM mainframe hardware and architecture; in particular when it comes to the topic of system virtualization using the z/VM operating system.

The simplified presentation of mainframe hardware and architecture will scarcely suffice in regard to practical operation of mainframes hardware, and to fully understand the complex z/OS operating system. It can, however, be a splendid starting point for further exploration.

But when recognising that z/VM and Linux hastily conquer and expand mainframe ground, the somewhat simplified view is easily justified. Unfortunately it leaves out some of the impressive reliability, availability and serviceability features (like Geographically Dispersed Parallel Sysplex, GPSP), which really singles out the mainframe platform.

According to thesis purpose and goals the report describes how the z/VM operating system is able establish virtual machines, which behave like real machines in almost every detail. The description is based on general section on virtualization theory, which also applies for non-mainframe platforms (e.g. virtualization using WMware and Xen on the x86 platform).

It is explained how z/VM exploits the mainframes hardware virtualization support to achieve very high virtualization efficiency; and how it offers “paravirtualization capabilities” or “hypervisor calls” to enhance performance even further for operating systems, which are “virtualization aware” (CMS in particular).

The ability of z/VM to share and distribute hardware resources is presented, taking a starting point in resources typically allocated to virtual Linux servers (“Linux guests”). Especially the measures to control and predict processor resource allocation is in focus. A relatively simple method (including a pseudo function) has been developed in other to give a good estimate of the distribution of processor power between individual virtual machines (and the virtual processors within these).

The theoretical assessment of the virtual environment is supported empirically, with a number of performance tests. These tests have provided hands-on experience, and a practical comprehension of the system. The performance tests given in report illustrate the behaviour of mainframe hardware, z/VM, Linux, and the combination of the three. These tests should generally contribute to a better understanding of running Linux under z/VM on the mainframe. Additionally the tests contribute with a few concrete performance optimization recommendations for the company specific setup.

The thesis finally touches upon the topic of availability of mainframe linux services – solely focusing on monitoring software according to company wishes. The monitoring chapter should only be regarded as an offset for further investigation, since it has not been possible to make thorough study of this topic in due time.

All in all the main goals for this thesis have been achieved, except when it comes to finding most suitable monitoring software solution for KMD. The thesis does however still provide the detailed platform knowledge, and initial monitoring considerations, which can useful in order to fulfil the goal. The thesis distinguishes itself from other work by:

- Providing a concise, yet theoretically and practically balanced, description of system virtualization on the mainframe using z/VM.
- Formulating a relatively simple method to estimate the distribution of processor resources between the individual virtual machines in z/VM.
- Supplying a number of developed test programs and tools, including:
    - A little, robust, multithreaded, and on-demand memory consuming, test program, which e.g. can estimate memory access rate and help reveal Linux swap plus z/VM paging performance issues.
    - A highly customizable solution to monitor practically any internal Linux values – using standard Linux components only.
    - A monitoring client program, which can log and plot monitored data on the run.
- Providing tangible test results for generally accepted performance optimization methods, while providing some insight in running z/VM and Linux.

Future work

From an academic point of view, it could would generally be wise to acknowledge the mainframe for the advantages it possesses; and to accept it as a current and future platform, since nothing but general misperceptions indicate the mainframe is about to vanish. Its single biggest threat is probably the lack of experienced mainframe software developers and operators; a problem, which IBM has responded to by arranging mainframe courses on universities around the world. It is however a miserable situation, if an extremely reliable, secure, cost competitive, and energy efficient platform actually dies, because of general ignorance and misperceptions.

As a concrete example it could be interesting to research whether any of the mainframe techniques (e.g. the processor dispatching and scheduling techniques in z/VM and PR/SM) could benefit system based on the emerging multi core processors. The mainframe has after all a head start as Symmetric MultiProcessing (SMP) system. Mainframe technology and software can on the other hand also be expected to benefit from findings in the world of distributed computing.

From company (KMD) perspective it would be logical to implement the recommendation from the performance tests (if not already done). The next step would be to finish the monitor software investigation and implement the best suited software.

If it later becomes necessary to tweak disk performance, it could be useful to test the influence of the different Linux I/O schedulers. It could also be useful to look at SCSI via fibre (FBA DASD devices), in order to make it easier to provide the large disks, which it lately has become more evident that the Perspektiv application requires.

Finally a Linux distribution upgrade from SLES9 SP3 to SLES10 SP1 should be considered. Mainly for the improvements in the Linux kernel, e.g. in regard to "steal time" but more importantly to exploit improvements in z/VM and hardware concerning better hardware virtualization support (e.g. QDIO assist technology like "QDIO Enhanced Buffer-State Management" and "Host Page-Management Assist, see p. 52).

BIBLIOGRAPHY

[1] Ibm system z partitioning achieves highest certification. http://www-03.ibm.com/systems/z/security/certification.html.

[2] Itil newsletter: Introducing itsm. http://itsm.the-hamster.com/itsm1.htm on 2007-09-26.

[3] Linux kernel (2.6.5-7.244) documentation: Proc filesystem. /usr/src/linux/Documentation/filesystems/proc.txt.

[4] Product information: How hcd and hcm help you. http://www-03.ibm.com/servers/eserver/zseries/zos/hcm/hcmhtmls/hcmpinfs% .html.

[5] Vm scheduler basics - virtual multiprocessor support. IBM z/VM homepage: VM performance tips http://www.vm.ibm.com/perf/tips/schedule.html.

[6] Ibm mainframe partitioning: Lpar vs vm. *Computer Economics Report*, 2001 23 11 4-8, November 2001.

[7] Ibm z/vm v5r1.0 cms user's guide. SC24-6079-00, http://publibz.boulder.ibm.com/epubs/pdf/hcsd7b00.pdf, September 2004.

[8] z/vm: Glossary, September 2004. GC24-6097-00.

[9] The term "swapcached" in /proc/meminfo. "kmerley's blog" on Kerneltrap.org, 2004-2005. http://kerneltrap.org/node/4097.

[10] Ibm tivoli software: Manage the broad range of systems that supports your high-priority applications. ftp://ftp.software.ibm.com/software/tivoli/brochures/bc-mng-brd-ra.pdf, 2005. GC28-8386-00.

[11] Ibm, z/architecture - principles of operation. SA22-7832-04, http://publibz.boulder.ibm.com/epubs/pdf/a2278324.pdf, September 2005.

[12] Program directory for z/vm. http://www.vm.ibm.com/progdir/zvm52000.pdf, December 2005. GI11-2860-00.

[13] z/os: Hardware configuration definition user's guide, September 2005. SC33-7988-05.

[14] z/vm: Guide for automated installation and service, December 2005. GC24-6099-02.

[15] z/vm: I/o configuration, v5r2. http://publibz.boulder.ibm.com/epubs/pdf/hcsg1b10.pdf, December 2005. SC24-6100-01.

[16] z/vm reference guide (v5r1). IBM, http://www-07.ibm.com/servers/eserver/includes/download/gm130137.pdf, January 2005. GM13-0137.

[17] z/vm: Service guide, December 2005. GC24-6117-01.

[18] z/vm: Vmses/e intoduction and reference, December 2005. GC24-6130-01.

[19] Certification report: Pr/sm lpar for the ibm system z9 enterprise class and the ibm system z9 business class. Bundesamt für Sicherheit in der Informationstechnik http://www.commoncriteriaportal.org/public/files/epfiles/0378a.pdf, September 2006.

[20] Dtu course 02337, lecture slides 2006-02-08. CampusNet, 2006.

[21] Ibm systems virtualization (r2v1). http://publib.boulder.ibm.com/infocenter/eserver/v1r2/topic/eicay/eicay% .pdf, September 2006.

[22] Ibm z/vm: General information, v5r2. GC24-6095-04, http://www.vm.ibm.com/pubs/hcsf8b11.pdf, April 2006.

[23] *Linux on System z, Device Drivers, Features, and Commands December, 2006*. International Business Machines Corporation 2000, 8 edition, December 2006. SC33-8281-03.

[24] Wikipedia: Mainframe computers. http://en.wikipedia.org/w/index.php?title=Mainframe_computer&oldid=1032% 22453, January 2006.

[25] z/os: Hardware configuration definition planning. http://publibz.boulder.ibm.com/epubs/pdf/iea2g870.pdf, September 2006. GA22-7525-10.

[26] z/vm: Connectivity, May 2006. SC24-6080-03.

[27] z/vm: Cp commands and utilities reference, May 2006. SC24-6081-03.

[28] z/vm: Cp planning and administration, May 2006. SC24-6083-03.

[29] z/vm: Cp programming services, May 2006. SC24-6084-02.

[30] z/vm: Performance, May 2006. SC24-6109-02.

[31] z/vm: Virtual machine operation, May 2006. SC24-6128-02.

[32] Esalps for managing z/linux and z/vm performance. http://velocitysoftware.com/whylps.html, August 2007.

[33] Esalps for managing z/linux and z/vm performance. http://www.velocity-software.com/esalps.htmll on 2007-09-26, September 2007.

[34] Hp-ux servers - npartitions. http://h20338.www2.hp.com/hpux11i/cache/323751-0-0-0-121.html, August 2007.

[35] Ibm system z webpage: Specialty engines. http://www-03.ibm.com/systems/z/specialtyengines/, February 2007.

[36] Ibm system z9 enterprise class (ec) reference guide, April 2007. http://www-03.ibm.com/systems/z/pdf/ZSO03005-USEN-01_z9EC_RefGuide.pdf.

[37] Ibm system z9 enterprise class update - frequently asked questions. http://www.ibm.com/common/ssi/fcgi-bin/ssialias?infotype=PM&subtype=RG&% appname=STG_ZS_USEN&htmlfid=ZSQ03014USEN&attachment=ZSQ03014USEN.PDF, April 2007.

[38] Ibm's project big green spurs global shift to linux on mainframe. IBM Press Release: http://www-03.ibm.com/press/us/en/pressrelease/21945.wss, August 2007.

[39] Kmd a/s: Annual report 2006. http://www.kmd.dk/aarsrapport, 2007.

[40] *Linux on System z, Device Drivers, Features, and Commands February, 2007*. International Business Machines Corporation 2000, February 2007. SC33-8289-03.

[41] mon - service monitoring daemon. http://mon.wiki.kernel.org/index.php/Main_Page on 2007-09-26, September 2007.

[42] Nagios homepage: About nagios. http://www.nagios.org/about/ on 2007-09-26, September 2007.

[43] Novell suse linux package description and support level information for contracted customers and partners. http://support.novell.com/products/server/supported_packages/SLES_10_s3% 90x_SP1.pdf, Juli 2007.

[44] Wikipedia: Linux on zseries. http://en.wikipedia.org/w/index.php?title=Linux_on_zSeries&oldid=10510% 4118, February 2007.

[45] Wikipedia: Virtualization. http://en.wikipedia.org/w/index.php?title=Virtualization&oldid=15252066% 7, August 2007.

[46] z/vm reference guide (v5r3). IBM, http://www.vm.ibm.com/library/zvmref3a.pdf, April 2007. ZSO03006-USEN-01.

[47] 2nd, editor. *z/VM: Getting Started with Linux on System z9 and zSeries*. IBM, December 2005. SC24-6096-01.

[48] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. *ACM SIGPLAN Notices, Proceedings of the 2006 ASPLOS Conference*, 41(11):2–13, 2006.

[49] Alan Altmark. z/vm security and integrity. http://www.vm.ibm.com/devpages/ALTMARKA/V71.pdf, April 2007.

[50] Bill Bitner. z/vm virtualization basics. slides, http://www.vm.ibm.com/devpages/bitner/presentations/virtualb.pdf, April 2005.

[51] Christian Borntraeger. Monitoring linux guests and processes with linux tools. http://linuxvm.org/Present/SHARE108/S9266cb.pdf , February 2006. SHARE.org 2006 session 9266.

[52] Christian Bornträger and Martin Schwidefsky. Providing linux 2.6 support for the zseries platform. *IBM Systems Journal, Vol 44, No 2, 2005*, 2005.

[53] David Boyes. Which linux distro is the best? TechTarget Expert Answer Center http://expertanswercenter.techtarget.com/eac/knowledgebaseAnswer/0,2951% 99,sid63_gci1141627,00.html on 2007-09-16, July 2005.

[54] Charlie Burns. The mainframe is dead: Long live the mainframe. Saugatuck Technology, Research Alert: http://research.saugatech.com/fr/researchalerts/364RA.pdf, July 2007. RA-364.

[55] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM Journal of Research and Development*, 25, issue 5:483–490, 1981.

[56] Denny Dutcavich. Best practices for oracle on linux for system z: Getting started. zSeries Oracle Special Interest Group Conference 2007, April 2007. http://zseriesoraclesig.org/2007presentations/Best_Practices_for_Oracle% .pdf.

[57] Mike Ebbers, Wayne O'Brian, and Bill Ogden. *Introduction to the New Mainframe: z/OS Basics*. International Business Machines Corporation 2005, January 2006. SG24-6366-0.

[58] John Fisher-Ogden. Hardware support for efficient virtualization. www.cse.ucsd.edu/ jfisherogden/hardwareVirt.pdf, ultimo 2006 (approx.). Research Exam by Ph.D. Student at Dept. of Computer Science & Engineering, University of California, San Diego.

[59] Gregory Geiselhart, Robert Brenneman, Eli Dow, Klaus Egeler, Torsten Gutenberger, Bruce Hayden, and Livio Sousa. *Linux for IBM System z9 and IBM zSeries*. IBM Redbooks, January 2006. SG24-6694-00, http://www.redbooks.ibm.com/abstracts/sg246694.html?Open.

[60] Gregory Geiselhart, Robert Brenneman, Torsten Gutenberger, Jean-Louis Lafitte, William Ventura, and Simon Williams. *Linux for zSeries: Fibre Channel Protocol Implementation Guide*. IBM Redbooks, August 2004. SG24-6344-00, http://www.redbooks.ibm.com/abstracts/sg246344.htmll.

[61] Gregory Geiselhart, Laurent Dupin, Deon George, Rob van der Heij, John Langer, Graham Norris, Don Robbins, Barton Robinson, Gregory Sansoni, and Steffen Thoss. *Linux on IBM eServer zSeries and S/390: Performance Measurement and Tuning*. IBM Redbooks, May 2003. SG24-6926-00, http://www.redbooks.ibm.com/abstracts/sg246926.html?Open.

[62] Klaus Johansen. Execution and monitoring of linux under z/vm, February 2007.

[63] Mike Kahn. The beginning of i.t. civilization – ibm's system/360 mainframe. *The Clipper Group - Captain's Log*, March 2004.

[64] Frank Kyne, Michael Ferguson, Tom Russell, Alvaro Salla, and Ken Trowell. *z/OS Intelligent Resource Director*. IBM Redbooks, August 2001. SG24-5952-00, http://www.redbooks.ibm.com/abstracts/sg245952.html?Open.

[65] Damian L. Osisek, Kathryn M. Jackson, and Peter H. Gum. Esa/390 interpretive-execution architecture, foundation for vm/esa. *IBM Systems Journal, Vol 30, No 1*, 1991.

[66] Lydia Parziale, Eli Dow, Klaus Egeler, Jason Herne, Clive Jordan, Edi Lopes Alves, Eravimangalath P. Naveen, Manoj S Pattabhiraman, and Kyle Smith. *Introduction to the New Mainframe: z/VM Basics*. IBM Redbooks, 1th, draft edition, August 2007. SG24-7316-00.

[67] Eberhard Pasch. Linux on system z performance hints & tips. SHARE Tampa Session 2591/9301 - http://download.boulder.ibm.com/ibmdl/pub/software/dw/linux390/perf/pht%_share_tampa_2007.pdf, March 2005.

[68] Keith Rochford, Brian Coghlan, and John Walsh. An agent-based approach to grid service monitoring. *ispdc*, 0:345–351, 2006. 10.1109/ISPDC.2006.

[69] Amit Singh. An introduction to virtualization.
http://www.kernelthread.com/publications/virtualization/, February 2004.

[70] BMC Software. Bmc® mainview® for linux - servers, datasheet.
http://www.bmc.com/products/documents/34/17/13417/13417.pdf, September 2002.

[71] Michael Steil. Inside vmware: How vmware, virtualpc and parallels actually work. 23rd Chaos Communication Congress, 2006.
http://events.ccc.de/congress/2006/Fahrplan/events/1592.en.html.

[72] Henrik Storner. About hobbit. http://hobbitmon.sourceforge.net/docs/about.html on 2007-09-26, September 2007.

[73] Darryl K. Taft. Maximizing the mainframe. eWeek, Enterprise News & Reviews:
http://www.eweek.com/article2/0,1895,2161658,00.asp, July 2007.

[74] Melinda Varian. Vm and the vm community: Past, present, and future. SHARE 89, Sessions 9059-9061, August 1997.

[75] Linus Vepstas. Homepage: Why port linux to the mainframe? http://linas.org/linux/i370-why.html, 2007-09-16, November 1999.

[76] Linus Vepstas. Homepage: Linux on the ibm esa/390 mainframe architecture.
http://www.linas.org/linux/i370-bigfoot.html, 2007-09-16, February 2000.

[77] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. *University of Washington Technical Report*, February 2002.

[78] Bill White, Roy Costa, Michael Gamble, Franck Injey, Giada Rauti, and Karan Singh. *HiperSockets Implementation Guide*. IBM Redbooks, March 2007. SG24-6816-01,
http://www.redbooks.ibm.com/abstracts/sg246816.html?Open.

[79] Bill White, Marian Gasparovic, and Dick Jorna. *IBM System z Connectivity Handbook*. IBM Redbooks, sg24-5444-07 edition, July 2007.

[80] Bill White, Franck Injey, Greg Chambers, Marian Gasparovic, Parwez Hamid, Brian Hatfield, Ken Hewitt, Dick Jorna, and Patrick Kappeler. *IBM System z9 Enterprise Class Technical Guide*. IBM Redbooks. IBM Redbooks, 3th edition, June 2007. SG24-7124-02,
http://www.redbooks.ibm.com/abstracts/sg247124.html?Open.

INDEX

APPENDIX A: PERFORMANCE TESTS

1. TEST 1: Number of processors per Linux guest

1.1 Measurements tools and configuration

1.1.1 Specialized monitoring from within Linux: qkumon

The developed xinetd service, qkumon (given in Appendix B.1), is used as the main monitoring tool during this test. More specifically this test uses test option "t", which invokes "top" and parses the about using the script "mon.awk" (see Appendix B.1.1 and B.1.4.1).

The qkumon java client is used to inquire the qkumon service from a remote host. The Java client plots the results in near real time view and logs data to a file for later data processing.

1.1.2 IBM Performance Toolkit for VM

The IBM Performance Toolkit is used to verify the general CPU usage and to supply the x'44 diagnose count. The Toolkit can be accessed via 3270 (z/VM guest: perfsvm) or as a web service on VM system IP address port 81.

A monitoring interval of 30 seconds is recommended to able to give a better picture of the workload behaviour over time. (Default setting is 60 sec.). From a privileged user run:

```
monitor sample interval nn sec
```

From within perfsvm user (if the necessary user class has been applied):

```
cp monitor sample interval nn sec
```

To store history files for the specific Linux guests for later analysis enable benchmarking for the specific guests. From within performance monitor (3270):

```
FC BENCH USER USERID FILE 00:00 TO 23:59
```

From script or activated from other user:

```
FCONCMD FC BENCH USER USERID FILE 00:00 TO 23:59
```

USERID should be replaced with the name of the VM guest in question and the benchmarking period can be adjusted fix local needs. History files can be analysed using Performance Toolkit option "32: History data files*" or "31: Graphics selection".

The X'44 Diagnose instruction count is derived from Toolkit option 4 "Privileged Operations". The website unfortunately has to be supervised and the value recorded manually.

1.2 WL1: Complete payroll processing

1.2.1 Workload description

WL1 is a KMD “Perspektiv Løn” specific workload. This job covers complete payroll processing (pdf and postscript file creation excluded). The print output stream is redirected to disk. This job estimated to typically run a couple of times before the final money transfers and account updates. Furthermore larger customers run this kind of job as a batch job each night to detect potential problems before final payroll processing.

The workload is of “mixed nature”: A combination CPU usage (application calculation + database queries) and disk usage (database, output file writing). Several of the application “sub” programs are called during the processing, which results in birth and termination of new processes, many of which accesses and manipulates the database.

1.2.2 Prerequisites

- Working “Perspektiv Løn” / mrdata application with adequate amount of test data (minimum 100 employees).
- Dummy printer definition redirecting print to plain file instead of printing using lpd or similar.

1.2.3 Settings and general test description

- Start-up the “Perspektiv” application and log in.
  (Linux user mrdata, run /MR/MrMenu/Menustart) .
- If necessary change the active company in Perspektiv: “sf” command
- Run application 410: “Total Lønafvikling” (Complete payroll processing)
- “Afl. form" (delivery form): Typically 20 (and/or 10, 21).
- Print to: p(rinter): dummy
- Press <F1> to approve...
- Functions: choose all (putting x’es) and destination printer (p, dummy).
- <F1> to approve and start processing.

1.3 WL2: Payslip PDF and PS file creation

1.3.1 Workload description

Like WL1 this workload is a KMD "Perspektiv Løn" specific workload. It covers complete payroll processing, but *includes* pdf and postscript file creation. This task is assumed to run a couple of times during the month: When pay slips are created for different payment intervals.

For a short period the workload begins like test 1: with a combination of CPU usage (application calculation + database queries) and disk usage (database, output file writing). The process is dominated by pdf-file creation and pdf to ps conversion, which is characterized by heavy CPU usage caused by a two sequentially running processes.

1.3.2 Prerequisites

- Working "Perspektiv Løn" / mrdata application with adequate amount of test data (minimum 100 employees).
- "Perspektiv Løn" PDF option enabled (compiled with "P" in char array "lonspec_dest" in MR.h. MR-script, MR.sh, updated in section "LONSPEC_PDF_MAIL" to delete ps-file and move pdf-file to tmp-directory).

1.3.3 Settings and general test description

- Start-up the "Perspektiv" application and log in.
  (Linux user mrdata, run /MR/MrMenu/Menustart) .
- If necessary change the active company in Perspektiv: "sf" command
- Run application 422: "Udskrifter: Lønspecifikationer" (Print: Pay slips)
- "Afl. form" (delivery form): Typically 20 (and/or 10, 21).
- Print to: p(rinter): pdf
- Press <F1> to approve and start processing.

1.4 Test results

1.4.1 Workload duration (run time)

		Guest 1	Guest 2	Time saved	saving	Performance toolkit CPU usage	
Test	Repetition	Linx02	Linx03	with 1 CPU	in %	Linx02	Linx03
		runtime	runtime	(sec.)		max	max
		(sec.)	(sec.)			%	%
T1.1	1	42	36	6	14%	82.4	82.3
	2	35	30	5	14%	61.7	60.6
	3	36	28	8	22%	90.9	89
	4	35	29	6	17%	60.6	56.7
T1.2	1	203	211	-8	-4%	98.1	96.2
	2	193	190	3	2%	98.4	97.6
	3	200	200	0	0%	99	97.6
	4	185	187	-2	-1%	99.3	97.8
T1.3	1	44				58.9	
	2	32				94.7	
T1.4	1		41				61.5
	2		28				85.9
T1.5	1	163				101	
	2	189				99.9	
T1.6	1		190				99.2
	2		191				99.3
T1.7	1	45	48	3	6%	55.5	64
	2	29	31	2	6%	90.5	91.1
	3	29	31	2	6%	93.2	89.8
	4	28	31	3	10%	89.7	90.6
T1.8	1	198	199	1	1%	98.2	98.2
	2	186	186	0	0%	97.4	98.3
	3	199	198	-1	-1%	97.3	98
	4	201	204	3	1%	97.2	97.8
T1.9	1	29				89.1	
	2	27				91.7	
	3	28				91.7	
T1.10	1		31				94.9
	2		32				92.5
	3		31				93.4
T1.11	1	201				99.4	
	2	190				99.5	
T1.12	1		204				101
	2		190				100

▨ Guest running with 1 CPU

Table A1-1: Test 1; Runtime and maximum CPU-usage (real) test results

1.4.2 Diagnose x'44 count

		x'44 / sec
		6.7
		11.6
		19.9
T1.3	WL1	3017
Linx02		2759
2 CPU		33.5
		18.3
	WL1	5291
		590
		11.4
		4.2
T1.4	WL1	4.1
Linx03		16.9
1 CPU		4.7
	WL1	14.2
		17.3
		42.7
		26.1
		46.3
		51
T1.5	WL2	1924
Linx02		3.7
2 CPU		2.7
		17.3
		15.5
		188
		85.3
		12
		6.1
		11.8
		14.2
		12.1
	WL2 (KGF	377
		59.4
		210
		6
		13.6
		201
	WL2 (KP)	1927
		7.9
		48.6
		94
		27.8
		89.2
		146
		6
		5.1
		6.7
		47.3
T1.6 (3)	WL2	0.1
Linx03		32.8
1 CPU		21.4
		1.3
		68
		48.4
		0.4
		2.4
		1
		0.6
		9.3
		329
		9.7
		16.8
T1.6 (3)	WL2	15.6
Linx03		0.3
1 CPU		1.7
		9.8
		8
		93.2
		9.8
		0.1
		6.4
		18.7
		5.4
T1.9	WL1	14.5
Linx02		30.7
1 CPU	WL1	615
		4.5
		4.5
		5.3
		4.6
	WL1	5.9
		13.9
		6.2
		17.7
T1.10	WL1	5367
Linx03		323
2 CPU	WL1	5365
		508
		28.2
		10.8
	WL1	5369
		443
		4.8
		5.7
		5.4
		6.7
		7.3
		9.5
		5.5
T1.9	WL1	6.1
Linx02		6.1
1 CPU		5.6
		4
		5.3
T1.11	WL2	5.7
Linx02		7
1 CPU		34.6
		30.8
		6.9
		5.6
		4.5
		2.9
		5.1
		19.4
T1.11	WL2	5.1
Linx02		6.4
1 CPU		5.1
		6.4
		6.5
		5.5
		3.7
		5.9
		11.1
		8.4
T1.12	WL2	41.7
Linx03		1768
2 CPU		8.6
		8.3
		25.8
		23.3
		127
		5.3
		28.4
	WL2	1737
		3
		6.7
		3.4
		4.6
		4
		307
		5.9
		6

Table A1-2: Diagnose x'44 instruction count readings from Performance Toolkit. Running workloads indicated by black rectangles.

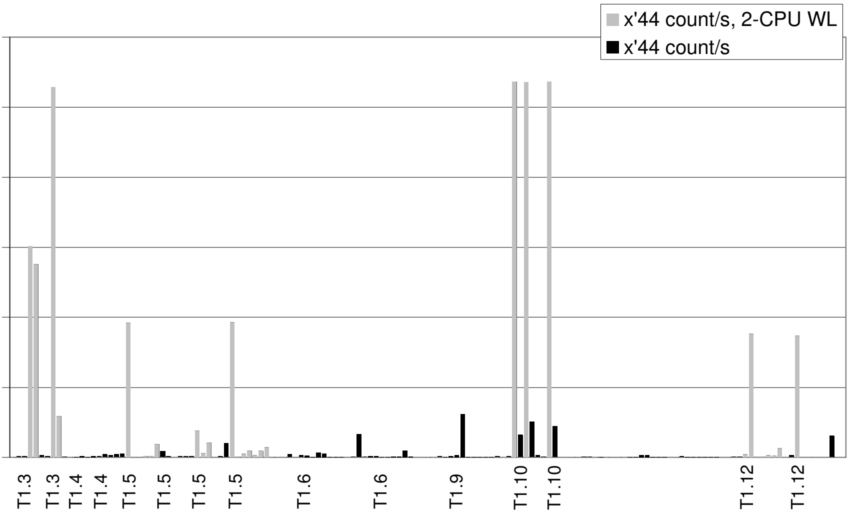


Figure A1-1 Diagnose x’44 instruction count over time. Gray bars indicate when a 2-CPU guest are processing workload. Black bars indicate all other counts (“back-ground noise” / idle time situations and when 1-CPU guest processes workloads). As expected 2-CPU guests issues many diagnose x’44’ instructions.

1.4.3 Workload characteristics

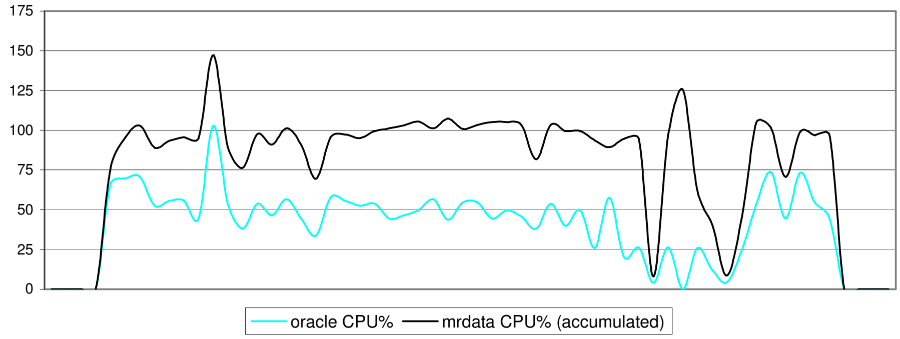


Figure A1-2: CPU usage characteristics for *WL1*. Contributions from oracle owned and mrdata (application) owned processes in accumulated view. Performance toolkit reading confirms general CPU readings ~ 1 whole CPU. (Not normalized Linux readings, T1.7, 2-CPU-guest, workload during/execution time: 31s).

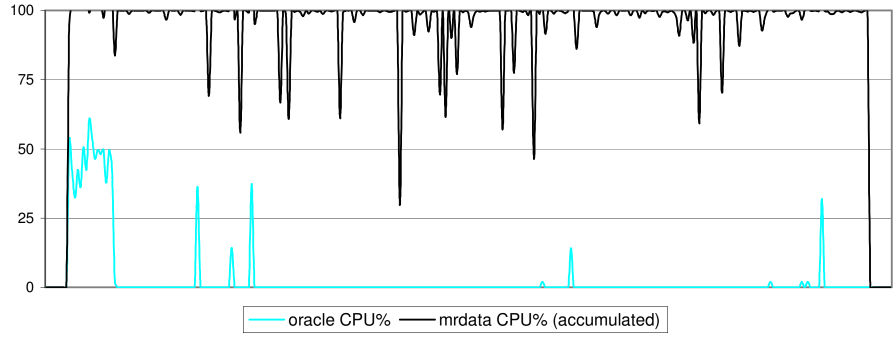


Figure A1-3: CPU usage characteristics for *WL2*. Contributions from oracle owned and mrdata (application) owned processes in accumulated view. Performance toolkit reading confirms general CPU readings ~ 1 whole CPU. (Not normalized Linux readings, T1.8, 2-CPU-guest, workload during/execution time: 198s).

2. TEST 2: Memory usage

All information on test 2 is provided in the report. This section is provided to keep maintain consistent numbering.

3. TEST 3: Disk I/O, LVM stripes and caches

3.1 Test results

3.1.1 Bonnie data from all tests (and repetitions)

	Sequential Write (nosync)						Sequential Read				Random seek	
Test no.&	Seq. Wr (char)		Seq. Wr (block)		Seq. Wr (rewrite)		Seq Rd (char)		Seq. Rd (block)		Rand. Rd	
repetition	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU	K/sec	%CPU
T3.1 R1	15649	99.1	138469	47	152236	35.7	15673	99.8	1065589	99.6	66204.5	116
T3.1 R2	15721	99.8	159941	54	39998	10	15664	99.9	1070200	100	60625.4	90.9
T3.1 R3	15692	99.7	123215	41.6	146013	32	15739	100	1062343	99.3	62647.8	94
T3.2 R1	15280	98.9	37659	13.6	51250	14.4	15586	99.9	1044292	99.6	90291.4	135
T3.2 R2	15759	99.7	40885	17.4	71241	19.8	15795	100	1058746	101	81637.7	102
T3.2 R3	16053	99.6	40236	16.6	54947	14.7	16097	99.9	1068024	99.8	59874.9	89.8
T3.3 R1	15714	99.7	31865	18.4	32764	14.1	14699	94.8	412219	75.5	26384.2	152
T3.3 R2	15224	99.1	28996	18.7	32375	11.9	14815	96.4	470583	88.9	15006.3	101
T3.3 R3	15619	99.3	42590	22.2	32404	12.4	15097	97.8	441478	81.7	25873.9	155
T3.4 R1	7336	48.9	11562	4.3	13201	3.7	14099	90.6	460521	74.7	28356.7	156
T3.4 R2	15424	98.5	32626	15.5	13508	3.7	14862	95.3	423915	72	29662.8	156
T3.4 R3	15528	98.7	32916	14.8	13588	3.5	14770	94.8	451571	79.2	8526.2	53.3
T3.5 R1	15682	99.5	44696	21.1	26321	9	14698	94.4	124241	23.5	2532	8.9
T3.5 R2	15554	99.5	44811	21.4	22885	9	15043	96.5	133565	18.6	2684.6	14.8
T3.5 R3	15663	99.3	44344	21.4	28269	8.3	15117	96.8	128900	22.6	2677.3	7.4
T3.6 R1	15743	99.88	43731	21.9	27505	7.9	14878	95.7	134007	21.2	2696.4	10.1
T3.6 R2	15674	99.4	44659	18.1	26692	6.9	14852	95.4	128302	19.3	2671.2	12.7
T3.6 R3	15664	99.7	44571	20.5	28190	9.4	14803	95.9	126337	17.3	2716.9	13.6
T3.7 R1	15490	98.4	45193	21.3	26911	10.1	14927	96.7	119300	19.6	2775.3	11.8
T3.7 R2	15519	99.3	44404	20.5	27530	8.8	14970	96.5	120347	19.1	2839.7	7.8
T3.7 R3	15646	99.3	44304	21.1	27356	8.7	14997	96.6	114494	18.1	2518.3	12
T3.8 R1	15376	97.6	31358	13.9	14256	4.5	15175	96.7	43642	5.1	1170	3.8
T3.8 R2	15691	99.4	31505	15.3	14850	4.5	15022	97.1	44945	4.6	1120	3.6
T3.8 R3	15574	98.5	31695	14.6	14886	4.1	15041	97	44993	6.3	1221.7	4
T3.9 R1	15467	98.1	31855	14.8	14165	4.2	15041	96.4	44663	5.5	1146.3	2.9
T3.9 R2	15543	98.6	32348	13.8	14221	4.3	15102	96.8	44606	4.8	1211.6	4.2
T3.9 R3	15594	98.8	31453	14.8	14832	4.7	15228	97.3	44204	5	1107.3	2.5
T3.10 R1	15357	98.1	30991	16.1	13867	4.8	15053	96.5	43918	5	1149.8	2.9
T3.10 R2	15242	98	13704	14.5	14300	4.6	15164	97	43989	6	1193.7	5.4
T3.10 R3	15387	97.4	30813	16.9	14720	4.4	15022	95.8	44939	5.2	868.3	2.4

3.1.2 Per Character sequential read/write data

Test number	useMem	Swappinesspiness	Minidisk cache	LVM stripes	CU cache	Seq.write (char)	%CPU	Seq. read (char)	%CPU
T3.1		60	x	x	x	15	99.5	15	99.9
T3.2	x	60	x	x	x	15	99.4	15	99.9
T3.3	x	0	x	x	x	15	99.4	15	96.3
T3.4	x	0	x		x	12	82.0	14	93.6
T3.5	x	0		x	x	15	99.4	15	95.9
T3.6	x	0		x		15	99.7	14	95.7
T3.7	x	0	x	x		15	99.0	15	96.6
T3.8	x	0	x			15	98.5	15	96.9
T3.9	x	0				15	98.5	15	96.8
T3.10	x	0			x	15	97.8	15	96.4

APPENDIX B: TEST SCRIPTS & PROGRAMS

1. qkumon - specialized monitoring within Linux

qkumon is a developed xinetd service. The mon.sh test script sends output from standard tools like "`top`" and system information from the `/proc/` file system to AWK-scripts (provided below) for processing into a simple comma strings. These scripts in combination with `xinetd` function as a simple server, which can be contacted by an arbitrary client via TCP/IP on port 5557.

1.1 /qkumon/mon.sh

```
#!/bin/bash
LINES=100
export LINES
while test 1; do
    read command
    case "$command" in
      (t) /usr/bin/top -b -n 1 | /qkumon/ora_mr.awk;;
      (m) /qkumon/memMeminfo.awk /proc/meminfo
          /qkumon/memVmstat.awk /proc/vmstat ;;
      (mi)/bin/date +"Test date: %d/%m/%Y %H:%M:%S"
          echo "time,sec.nano,app,slab,PageTables,vmallocUsed,Buffers,
               Cached,MemFree,SwapCached,SwapEvictedFromRealMem,
               SwapFree,Committed_AS,Mapped,Active,Inactive,pgfault,
               pgmajfault,pswpin,pswpout";;
      (*) break;;
    esac
done
```

Be aware the string provided after the echo statement should be given on one line.

1.2 /etc/xinetd.d/qkumon

```
service qkumon
{
    socket_type   = stream
    protocol      = tcp
    wait      = no
    user      = root
    group        = root
    server       = /qkumon/mon.sh
    port      = 5557
}
```

1.3 Configuration and activation

In order to make the server functionality work, `xinetd` has to be installed and the "qkumon service" has to be specified in the "`/etc/services`" file:

```
qkumon  5557/tcp  # Simple script based monitoring tool
```

Afterward `xinetd` should be restarted:

```
/etc/init.d/xinetd restart
```

1.4 qkumon help scripts

1.4.1 /qkumon/ora_mr.awk

This awk scripts is hard coded to monitor “mrdata” and “oracle” owned processes in particular.

```
#!/usr/bin/awk -f
# AWK program
# Output: comma-separated values:
# oracle processes: Running, sleeping, undisruptable sleep, memory, cpu
# mrdata processes: running, sleeping, undisruptable sleep, memory, cpu
# rest processes: number running, cpu used

BEGIN {
        TIME="/bin/date +'%d/%m/%Y %H:%M:%S',%s"
        oRun=0; oSle=0; oDis=0;
        mRun=0; mSle=0; mDis=0;
        oMem=0; oCpu=0; mMem=0;
        mCpu=0; restCpu=0; restRunning=0
}

function Time() {
        TIME | getline t
        close(TIME)
        return t
}

($2=="oracle")&&($8=="R") {oRun++ ; oCpu+=$9 ; oMem+=$10 }
($2=="oracle")&&($8=="S") {oSle++ ; oCpu+=$9 ; oMem+=$10 }
($2=="oracle")&&($8=="D") {oDis++ ; oCpu+=$9 ; oMem+=$10 }
($2=="mrdata")&&($8=="R") {mRun++ ; mCpu+=$9 ; mMem+=$10 }
($2=="mrdata")&&($8=="S") {mSle++ ; mCpu+=$9 ; mMem+=$10 }
($2=="mrdata")&&($8=="D") {mDis++ ; mCpu+=$9 ; mMem+=$10 }
($9>0){restCpu+=$9}
($8=="R"){restRunning++}
END {
        restCpu -= (oCpu + mCpu)
        restRunning -= (oRun + mRun)
        print Time() "," oRun "," oSle "," oDis "," oMem "," oCpu ","
mRun "," mSle ","  mDis "," mMem "," mCpu "," restRunning "," restCpu
}
```

1.4.2 /qkumon/memMeminfo.awk

This script processes `/proc/meminfo` data into a comma separated format optimized for easy plotting.

```
#!/usr/bin/awk -f

BEGIN {
   TIME="/bin/date +'%H:%M:%S, %s.%N'"
}

function Time() {
   TIME | getline t
   close(TIME)
   return substr(t, 1, 10) substr(t, 14, 10)
}

# /^keyword/ => Searches for keyword in beginning of line
/^Slab/     {Slab=$2}
/^MemTotal/    {MemTotal=$2}
/^SwapCached/  {SwapCached=$2}
/^PageTables/  {PageTables=$2}
/^VmallocUsed/    {VmallocUsed=$2}
/^MemFree/     {MemFree=$2}
/^Buffers/     {Buffers=$2}
/^Cached/      {Cached=$2}
/^SwapTotal/   {SwapTotal=$2}
/^SwapFree/    {SwapFree=$2}
/^SwapCached/  {SwapCached=$2}
/^Committed_AS/ {Committed_AS=$2}
/^Mapped/      {Mapped=$2}
/^Active/      {Active=$2}
/^Inactive/    {Inactive=$2}

END {
   SwapEvictedFromRealMem  = SwapTotal - SwapFree - SwapCached
   apps = MemTotal-MemFree-Buffers-Cached-SwapCached-Slab-PageTables-
VmallocUsed
   printf ("%s,", Time())
   printf ("%d,", apps)
   printf ("%d,", Slab)
   printf ("%d,", PageTables)
   printf ("%d,", VmallocUsed)
   printf ("%d,", Buffers)
   printf ("%d,", Cached)
   printf ("%d,", MemFree)
   printf ("%d,", SwapCached)
   printf ("%d,", SwapEvictedFromRealMem)
   printf ("%d,", SwapFree)
   printf ("%d,", Committed_AS)
   printf ("%d,", Mapped)
   printf ("%d,", Active)
   printf ("%d,", Inactive)
}
```

1.4.3 /qkumon/memVmstat.awk

This script parses the `/proc/vmstat` file and extracts information on page faults and the number of pages swapped in and out, into a comma separated string.

```
#!/usr/bin/awk -f

# /^keyword/ => Searches for keyword in beginning of line
/^pgfault/     {pgfault=$2}
/^pgmajfault/  {pgmajfault=$2}
/^pswpin/      {pswpin=$2}
/^pswpout/     {pswpout=$2}


END {
   printf ("%d,", pgfault)
   printf ("%d,", pgmajfault)
   printf ("%d,", pswpin)
   printf ("%d\n", pswpout)
}
```

2. useMem - memory consumer program

2.1 useMem.c

```
// -------------------------------------------------------------------------
// * useMem.c - Memory hungry program e.g. for tests of swapping
// *
// * Startup argument: Memory to occupy from beginning in MB
// *
// * By Klaus Johansen (qku / s053075) 2007-07-19)
// *
// * Changelog:
// *   2007-07-20: Support of 0 worker threads + initialize increments
// *               added monitor thread
// *   2007-07-23: Thread safe wake-up of main after mem in/decrease
// *   2007-08-01: Renamed to useMem
// *   2007-08-20: Write comma separated, reset 'oldValues' on create thread
// *             added begin date, time stamp, etc.; messages to stderr
// *   2007-09-22: Corrected error: changed assignment to expression in if,
// *               and throughput calculation correspondingly
// -------------------------------------------------------------------------

#include <stdio.h>      // I/O...
#include <pthread.h> // Thread support
#include <time.h>       // Nanosleep
#include <stdlib.h>     // rand()
#include <errno.h>      // evaluate errors from nanosleep

#include <sys/time.h>   // for writeTime() including milli sec
#include <time.h>       // for writeDate()


#define MAX_THREAD 50
#define MAX_MEM (RAND_MAX)
#define MAX_MEM_MB (RAND_MAX/(1024*1024))
#define TOINC 0x04000000
#define _MULTI_THREADED

const unsigned int kb128iMax = 128*1024/sizeof(unsigned long int);

// -------------------------------------------------------------------------
// Prototyping
// -------------------------------------------------------------------------
void memChange(int);
void *worker(void *);
void *monitor(void *arg);
void threadChange(int);
void writeTime();
void writeDate();

// -------------------------------------------------------------------------
// User defined types
// -------------------------------------------------------------------------
typedef struct {
   int id;
   unsigned long int kb128count;
   unsigned int stop;
} workerParm;

// -------------------------------------------------------------------------
// Global variables
// -------------------------------------------------------------------------
unsigned int currentMemSize = 0x04000000;   // 64MB
unsigned int numActiveThreads = 1;

unsigned char wait = 0;
unsigned char stop = 0;
unsigned char flagToStop = 0;
unsigned char threadsWaiting = 0;
unsigned long int *memChunkPtr;
unsigned long int oldValues[MAX_THREAD];

struct timespec interval;  // timespec struc for nanosleep

// Working threads vars:
pthread_t *threads;
pthread_t monitorThread;
workerParm *parms;

// Sync related
pthread_mutex_t  count_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t      count_threshold_cv = PTHREAD_COND_INITIALIZER;

pthread_mutex_t  canRunAgain_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t      canRunAgain_cv = PTHREAD_COND_INITIALIZER;

// ------------------------------------------------------------------------
//  Main function
// ------------------------------------------------------------------------
int main(int argc, char* argv[]) {
    unsigned int i;

    // Initialization

    srand(5); // Seeding random number generator

    threads=(pthread_t *)malloc(MAX_THREAD*sizeof(*threads));
    parms=(workerParm *)malloc(sizeof(workerParm)*MAX_THREAD);

    interval.tv_sec = 0;
    interval.tv_nsec = 0; // 1000; // (long)(100*1e+6);

    // "Welcome message"
    fprintf(stderr, "Maximum memory usage: %dMB\n", MAX_MEM_MB);

    // Change memory size to occupy from start if argument is given
    if (argc > 1) {
        unsigned int tmp;
        if ( sscanf(argv[1], "%d", &tmp) == 1) {
            if (tmp*1024 <= RAND_MAX/1024){
                currentMemSize = 1024*1024*tmp;   // from MB to bytes
                fprintf(stderr, "Memory usage set to: %d bytes (%d MB)\n", currentMemSize,
                    currentMemSize/(1024*1024));
            } else {
                fprintf(stderr, "Argument not accepted.\n");
                exit(1);
            }
        }
        if (argc == 3) {
            if ( sscanf(argv[2], "%d", &tmp) == 1) {
                if (tmp <= MAX_THREAD){
                    numActiveThreads = tmp;
                } else {
                    fprintf(stderr,
                        "Argument 2 (number of threads) not accepted, should be < %d.\n" ,
                         MAX_THREAD);
                    exit(1);
                }
            }
        }
    } else {
        fprintf(stderr, "Memory usage defaluts to %d MB and number of threads to %d\n",
            currentMemSize/(1024*1024), numActiveThreads);
    }

    memChunkPtr = (unsigned long int *)malloc( currentMemSize );

    if (memChunkPtr == NULL) {
        fprintf(stderr, "Malloc failed, exiting...\n");
        exit(2);
    }

    memset( memChunkPtr, '\xAA', currentMemSize );
    fprintf(stderr, "Memory initialized\n");

    // Output Descriptive text and running date
    writeDate();
    printf ("time , elapsed time, threads, memory usage, thoughput (kb) \n");

    // Creation of workerthread
    for (i=0; i < numActiveThreads ; i++){
        int ret;

        parms[i].id=i;
        parms[i].stop = 0;
        parms[i].kb128count = 0;

        ret = pthread_create(&threads[i], NULL, worker, (void *)(parms+i));

        if ( ret== 0) {
            fprintf(stderr, "Thread %d created successfully\n", i);
        } else {
            fprintf(stderr, "Thread %d NOT created\n", i);
        }
    }

    // Creation of monitor thread
    {
        int ret = pthread_create(&monitorThread, NULL, monitor, NULL);
        if ( ret== 0) {
            fprintf(stderr, "Monitor thread %d created successfully\n", i);
        } else {
            fprintf(stderr, "Monitor thread %d NOT created\n", i);
        }
    }

    fprintf(stderr, "%d thread(s) running\n", numActiveThreads);

    // Interactive input loop;
    while (stop == 0){

        int cmd;
        cmd = getchar();

        switch (cmd) {
            case '-':
            case '+':
                memChange(cmd);
                break;
            case 't':
            case 'g':
                threadChange(cmd);
                break;
            case 'q':
                stop = 1;
                break;
        }
    }
    // Wait for all threads to stop
    for (i=0; i<numActiveThreads; i++) {
        pthread_join(threads[i],NULL);
    }

    // Free memory and exit
    free (memChunkPtr);
    return 0;

}
```

```
// -------------------------------------------------------------------------
// Function: memChange - Increase / decrease memory by 64MB
// input:    '+' increase memory
//         '-' decrease memory
// -------------------------------------------------------------------------
void memChange(int cmd){

   long int newMemSize=0;

   //threadsWaiting = 0;      // Reset number of waiting threads (from last run)
   flagToStop = 1;         // Signal threads to sleep as they reach end of loop

   // Wait for all active threads sleeping...
   pthread_mutex_lock(&count_mutex);
   while (threadsWaiting < numActiveThreads) {
      //pthread releases mutex while waiting...
      pthread_cond_wait(&count_threshold_cv, &count_mutex);
   }
   pthread_mutex_unlock(&count_mutex);

   if (cmd == '-') {
      //Decrease memory size
      newMemSize = currentMemSize - TOINC;
      if (newMemSize <= TOINC) { // > inc/dec block size
         fprintf(stderr, "New Value below %d MB\n", TOINC/1024/1024);
         newMemSize = 0;
      }
   } else {
      //Increase memory size
      newMemSize = currentMemSize  + TOINC;
      if (newMemSize > MAX_MEM) {
         fprintf(stderr, "New Value limited reached (%d MB)\n", MAX_MEM_MB);
         newMemSize = 0;
      }

   }
   // Resize memory chunk
   if (newMemSize > 0) {
      memChunkPtr = (unsigned long int *) realloc( memChunkPtr, newMemSize);
      currentMemSize = newMemSize;

      if (cmd == '+' ) {
         memset( ((unsigned char*)memChunkPtr)+currentMemSize-TOINC, '\xAA', TOINC );
         fprintf(stderr, "New memory initialized. ");
      }
      fprintf(stderr, "Memory usage: %d MB\n", currentMemSize/1024/1024);
   }

   // Signal all threads to start again... (safely)
   pthread_mutex_lock(&canRunAgain_mutex);
   flagToStop = 0;
   pthread_cond_broadcast(&canRunAgain_cv);
   pthread_mutex_unlock(&canRunAgain_mutex);

   // Wait for all active threads is awake again, since this function
   // should not be invoked again before all working threads is awake
   pthread_mutex_lock(&count_mutex);
   while (threadsWaiting > 0 ) {
      //pthread releases mutex while waiting...
      pthread_cond_wait(&count_threshold_cv, &count_mutex);
   }
   pthread_mutex_unlock(&count_mutex);

}
// -------------------------------------------------------------------------
// Function: threadChange - Increase / decrease number of worker threads
// input:    '+' increase memory
//         '-' decrease memory
// -------------------------------------------------------------------------
void threadChange(int cmd) {
   if (cmd == 't') {
   // Create a thread

      if (numActiveThreads < MAX_THREAD) {
         int ret;
         int i=numActiveThreads; // acutal count equals new array index

         parms[i].id=i;
         parms[i].stop = 0;
         parms[i].kb128count = 0;
         oldValues[i] = 0;

         ret = pthread_create(&threads[i], NULL, worker, (void *)(parms+i));

         if ( ret== 0) {
             numActiveThreads++;
             fprintf(stderr, "%d thread(s) running\n", numActiveThreads);
         } else {
             fprintf(stderr, "Thread %d NOT created\n", i);
         }
      } else {
         printf ("Maximum number of threads reached (%d)\n" , MAX_THREAD);
      }

   } else {
      // stop a thread
      if (numActiveThreads > 0) {
         parms[numActiveThreads-1].stop = 1;
         pthread_join(threads[numActiveThreads-1],NULL); // Await thread stopping
         numActiveThreads--;
         fprintf(stderr, "%d thread(s) running\n", numActiveThreads);

      } else {
         printf ("All threads already stopped\n");
      }
   }
}

// --------------------------------------------------------------------------
// Function: Worker thread - continuesly writes into memory chunck
// --------------------------------------------------------------------------
void *worker(void *arg) {

   workerParm *p=(workerParm *)arg;
   unsigned int actualIndex;
   unsigned kb128LocalCount = 0;

   struct timespec rest;

   // Find a random place in memory chunk where to begin iterations
   do {
      actualIndex = rand();
   }
   while ( actualIndex >= currentMemSize/sizeof(unsigned long int) );

   fprintf(stderr, "Workerthread %d has started with memory chunk index %d \n",
      p->id, actualIndex);

   // Repeat until thread is stopped...
   while ( p->stop == 0 && stop == 0) {

      // Increment iterator and fix potential overflow
      // Always do before accessing memory, since memsize might have changed
      actualIndex ++;
      if (actualIndex >= currentMemSize/sizeof(unsigned long int))
         actualIndex = 0;

      // Modify data in memory to ensure memory page in memory
      memChunkPtr[actualIndex] += kb128LocalCount;

      kb128LocalCount++;
      if (kb128LocalCount == kb128iMax) {
         kb128LocalCount= 0;
         p->kb128count++;
      }


      // Wait according to actual "interval"-interruption by a non-blocked
      // signal is handled
      while ( wait && nanosleep(&interval, &rest) == -1) {
         if (errno != EINTR ) {
             printf ("problem calling nanosleep in thread %d\n", p->id);
             exit(3);
         }
      }

      // Going to sleep if "main-thread" has flag'ed to do...
      if (flagToStop > 0) {

         // Signal main thread that thread is going to wait (safe manner)
         pthread_mutex_lock(&count_mutex);
         threadsWaiting ++;
         pthread_cond_signal(&count_threshold_cv);
         pthread_mutex_unlock(&count_mutex);

         //wait for main has finished / flagToStop turn 0 (safe manner)
         pthread_mutex_lock(&canRunAgain_mutex);
         while (flagToStop > 0) {
             //pthread releases mutex while waiting...
             pthread_cond_wait(&canRunAgain_cv, &canRunAgain_mutex);
         }
         pthread_mutex_unlock(&canRunAgain_mutex);

         // Signal main thread that this thead is running again (safe manner)
         pthread_mutex_lock(&count_mutex);
         threadsWaiting --;
         pthread_cond_signal(&count_threshold_cv);
         pthread_mutex_unlock(&count_mutex);

      }
   }

   fprintf(stderr, "Workerthread %d stopping\n", p->id);

   pthread_exit(NULL);

}
// ------------------------------------------------------------------------
// Function: Monitor thread -
//      calculates number of bytes processes by worker threads
//      Runs approximately 1 time pr. sec.
// ------------------------------------------------------------------------
void *monitor(void *arg) {

   unsigned long int sinceLast;
   unsigned int i;


   struct timespec interval;
   struct timespec rest;

   interval.tv_sec = 1;
   interval.tv_nsec = 0;

   // Init oldValues array
   for (i=0; i<MAX_THREAD; i++)
      oldValues[i] = 0;

   // Keep running until program is stopped
   while (stop == 0) {

      sinceLast=0; // reset counter
      // Since parms[i] is only updated by the threads, this is reasonably safe
      for (i=0; i<numActiveThreads; i++) {
         sinceLast += parms[i].kb128count - oldValues[i] ;
         oldValues[i]=parms[i].kb128count;
      }

      writeTime();
      printf("%d, %d, %d \n", numActiveThreads, currentMemSize/1024/1024, (sinceLast/8));


      // Wait according to actual "interval" - interruption by a non-blocked
      // signal is handeld
      while ( nanosleep(&interval, &rest) == -1) {
         if (errno != EINTR ) {
             printf ("problem calling nanosleep from monitor thread\n");
             exit(4);
         }
      }
   }

   pthread_exit(NULL);

}

// ---------------------------------------------------------------------------
// Function: writeTime -
//      Writes current time and program elapsed in format:
//      "hh:mm:ss , elapSec.elapMilliSec , "
// ---------------------------------------------------------------------------
void writeTime() {

   static time_t startSec = 0;

   struct timeval tv;
    struct timezone tz;
    struct tm *tm;

    gettimeofday(&tv, &tz);
    tm=localtime(&tv.tv_sec);

   if (startSec == 0)
      startSec = tv.tv_sec;

    printf("%d:%02d:%02d , %d.%d ,", tm->tm_hour, tm->tm_min, tm->tm_sec,
             tv.tv_sec-startSec, tv.tv_usec/1000);
}


void writeDate() {
   time_t rawtime;
   struct tm * timeinfo;

   time ( &rawtime );
   timeinfo = localtime ( &rawtime );
   printf ( "Test date: %s", asctime (timeinfo) );
}
```

2.2 Makefile for useMem (uM)

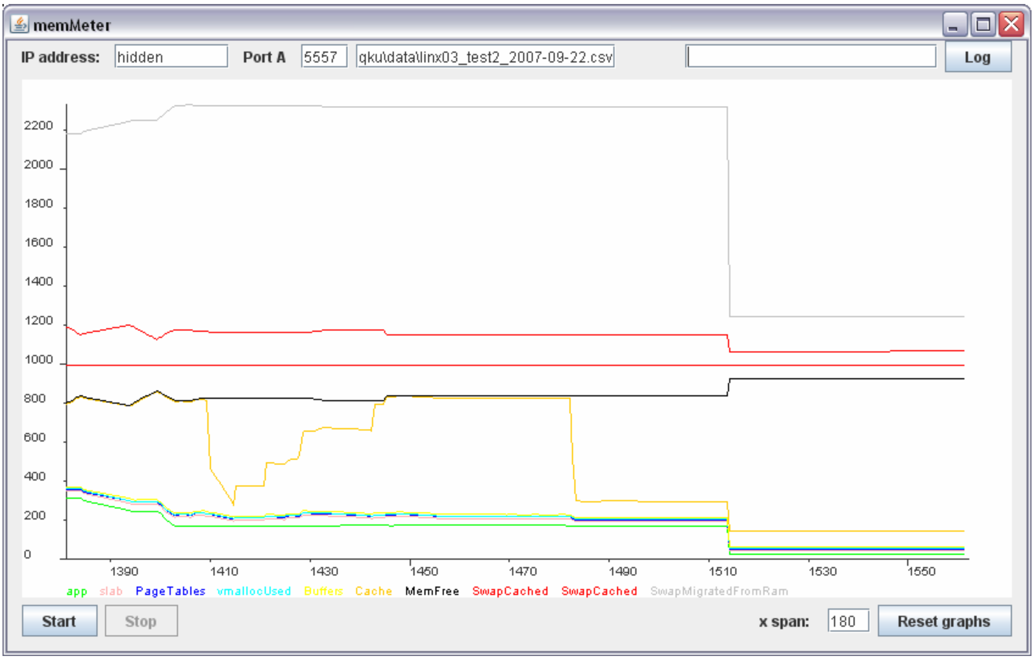
```
CC = gcc
LLIBS = pthread

all: useMem.c
    $(CC) useMem.c -lpthread -o uM -O3 -march=z900 -funroll-loops
```

3. memMeter, qkumon Java client

The memMeter program is a small Java program (see Figure 3-1) made in NetBeans. It establishes a TCP connection to the qkumon xinetd service mentioned above (B.1); it plots the received data and saves data in a log file as well.

The program uses the JChart2D charting library in version 2.2.1, which can be found via the project homepage: http://jchart2d.sourceforge.net/. It has been necessary to make some adjustments to one of existing charts (or more specifically one of the so-called “traces”), in order to produce a plot with a fixed interval on the x-axis. The source code for the main application class and the new “trace” are given below.



IP address: hidden
Port A 5557
qku\data\linx03_test2_2007-09-22.csv
x span: 180

Figure 3-1: The GUI of the MemMeter java client for the qkumon/mem.awk monitor.

3.1 qkuMeterGUI.java

```
import info.monitorenter.gui.chart.Chart2D;
import info.monitorenter.gui.chart.rangepolicies.RangePolicyForcedPoint;
import info.monitorenter.gui.chart.traces.Trace2DXspan;
import java.awt.Color;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.DataOutputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;
import java.net.UnknownHostException;
import java.util.Vector;
import javax.swing.JLabel;
import javax.swing.JTextField;

/*
 * qkuMeterGUI.java
 * Created on 23 May 2007, 14:54
 */

public class qkuMeterGUI extends javax.swing.JFrame {

    /**
     * Creates new form qkuMeterGUI
     */
    public qkuMeterGUI() {
        initComponents();
        // Create an ITrace:
        // Note that dynamic charts need limited amount of values!!!

        chartA.getAxisY().setRangePolicy(new RangePolicyForcedPoint() );
        meterThreadA = new dataGatherThread(chartA, textPortA);
    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    // <editor-fold defaultstate="collapsed" desc=" Generated Code ">
    private void initComponents() {
        jLabel1 = new javax.swing.JLabel();
        textIPaddress = new javax.swing.JTextField();
        buttonStart = new javax.swing.JButton();
        buttonStop = new javax.swing.JButton();
        buttonReset = new javax.swing.JButton();
        jLabel4 = new javax.swing.JLabel();
        textPortA = new javax.swing.JTextField();
        chartA = new info.monitorenter.gui.chart.Chart2D();
        jLabel6 = new javax.swing.JLabel();
        textXspan = new javax.swing.JTextField();
        textLogFile = new javax.swing.JTextField();
        buttonLog = new javax.swing.JButton();
        textLog = new javax.swing.JTextField();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("memMeter");
        jLabel1.setText("IP address:");

        textIPaddress.setText("172.31.218.12");
        textIPaddress.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                textIPaddressActionPerformed(evt);
            }
        });

        buttonStart.setText("Start");
        buttonStart.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonStartActionPerformed(evt);
            }
        });

        buttonStop.setText("Stop");
        buttonStop.setEnabled(false);
        buttonStop.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonStopActionPerformed(evt);
            }
        });

        buttonReset.setText("Reset graphs");
        buttonReset.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonResetActionPerformed(evt);
            }
        });

        jLabel4.setText("Port A");

        textPortA.setText("5557");
        textPortA.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                textPortAActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout chartALayout = new javax.swing.GroupLayout(chartA);
        chartA.setLayout(chartALayout);
        chartALayout.setHorizontalGroup(
            chartALayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 768, Short.MAX_VALUE)
        );
        chartALayout.setVerticalGroup(
            chartALayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGap(0, 425, Short.MAX_VALUE)
        );

        jLabel6.setText("x span: ");

        textXspan.setText("30");

        textLogFile.setText("c:\\qku\\data\\log.csv");

        buttonLog.setText("Log");
        buttonLog.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonLogActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, lay-
out.createSequentialGroup()
                .addContainerGap()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAILING)
                    .addComponent(chartA, javax.swing.GroupLayout.Alignment.LEADING,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.DEFAULT_SIZE,
Short.MAX_VALUE)
                    .addGroup(layout.createSequentialGroup()
                        .addComponent(jLabel1)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
```

```
                        .addComponent(textIPaddress, javax.swing.GroupLayout.DEFAULT_SIZE,
94, Short.MAX_VALUE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(jLabel4)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(textPortA, javax.swing.GroupLayout.PREFERRED_SIZE,
39, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(textLogFile, javax.swing.GroupLayout.PREFERRED_SIZE,
213, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addGap(57, 57, 57)
                        .addComponent(textLog, javax.swing.GroupLayout.PREFERRED_SIZE, 207,
javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(buttonLog))
                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING, lay-
out.createSequentialGroup()
                        .addComponent(buttonStart)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(buttonStop)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, 469, Short.MAX_VALUE)
                        .addComponent(jLabel6)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(textXspan, javax.swing.GroupLayout.PREFERRED_SIZE,
35, javax.swing.GroupLayout.PREFERRED_SIZE)

.addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(buttonReset)))
                .addContainerGap())
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(jLabel1)
                    .addComponent(textLogFile, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(buttonLog)
                    .addComponent(textLog, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(jLabel4)
                    .addComponent(textPortA, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                    .addComponent(textIPaddress, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(chartA, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)

.addGroup(layout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASELINE)
                    .addComponent(buttonStop)
                    .addComponent(buttonStart)
                    .addComponent(buttonReset)
                    .addComponent(jLabel6)
                    .addComponent(textXspan, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addContainerGap())
        );
        pack();
    }// </editor-fold>

    private void buttonLogActionPerformed(java.awt.event.ActionEvent evt) {
        if (bw != null) {
            try {
                bw.newLine();
                bw.write(textLog.getText());
            } catch (IOException ignore) {
                System.out.println("IOException writing log text");
            }
        } else {
            try {
                BufferedWriter bw = new BufferedWriter(
       new FileWriter(textLogFile.getText(),true));
                bw.newLine();
                bw.write(textLog.getText());
                bw.close();
            } catch (IOException ex) {
                System.out.println(
       "IOException writing log text (file not already open)\n" + ex);
            }

        }
    }

    private void textPortAActionPerformed(java.awt.event.ActionEvent evt) {
    }

    private void buttonResetActionPerformed(java.awt.event.ActionEvent evt) {
        oRunTrace.removeAllPoints();
        oSleTrace.removeAllPoints();
    }

    private void buttonStopActionPerformed(java.awt.event.ActionEvent evt) {

        buttonStart.setEnabled(true);
        buttonStop.setEnabled(false);

    }

    private void buttonStartActionPerformed(java.awt.event.ActionEvent evt) {
        buttonStart.setEnabled(false);
        buttonStop.setEnabled(true);
        meterThreadA.wakeup();
    }

    private void textIPaddressActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new qkuMeterGUI().setVisible(true);
            }
        });
    }

    // Variables declaration - do not modify
    private javax.swing.JButton buttonLog;
    private javax.swing.JButton buttonReset;
    private javax.swing.JButton buttonStart;
    private javax.swing.JButton buttonStop;
    private info.monitorenter.gui.chart.Chart2D chartA;
    private javax.swing.JLabel jLabel1;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JTextField textIPaddress;
    private javax.swing.JTextField textLog;
    private javax.swing.JTextField textLogFile;
    private javax.swing.JTextField textPortA;
    private javax.swing.JTextField textXspan;
    // End of variables declaration
```

```
dataGatherThread meterThreadA = null;
dataGatherThread meterThreadB = null;
Trace2DXspan oRunTrace = null;
Trace2DXspan oSleTrace = null;
Double xFirst = Double.MAX_VALUE;
BufferedWriter bw = null;

class dataGatherThread extends Thread {

    Vector<Trace2DXspan> traces = new Vector<Trace2DXspan>();
    JTextField textPort;
    JLabel labelCurrentVal;

    public Trace2DXspan createTrace(int parmnum, Color c, String name, String unit1,
   String unit2 ){
        Trace2DXspan trace = new Trace2DXspan();
        trace.setColor(c);
        trace.setName(name);
        trace.setPhysicalUnits(unit1, unit2);

        if (traces.size() < parmnum+1) {
            traces.setSize(parmnum+5);
        }
        traces.setElementAt(trace, parmnum);
        return trace;
    }

    public dataGatherThread(Chart2D chart, JTextField p) {

        traces.setSize(20);
        chart.addTrace(createTrace(2, Color.GREEN, "app", "", ""));
        chart.addTrace(createTrace(3, Color.PINK, "slab", "", ""));
        chart.addTrace(createTrace(4, Color.BLUE, "PageTables", "", ""));
        chart.addTrace(createTrace(5, Color.CYAN, "vmallocUsed", "", ""));
        chart.addTrace(createTrace(6, Color.YELLOW, "Buffers", "", ""));
        chart.addTrace(createTrace(7, Color.ORANGE, "Cache", "", ""));
        chart.addTrace(createTrace(8, Color.BLACK, "MemFree", "", ""));
        chart.addTrace(createTrace(9, Color.RED, "SwapCached", "", ""));
        chart.addTrace(createTrace(20, Color.RED, "SwapCached", "", ""));
        chart.addTrace(createTrace(10, Color.LIGHT_GRAY, "Swap", "", ""));

        textPort = p;
        start();
    }

    public synchronized void wakeup() {
        this.notify();
    }

    public synchronized void run() {

        //System.out.println("Thread is living!!!!!!!!");

        while(true) {
            Socket clientSocket = null;
            DataOutputStream os = null;
            BufferedReader br = null;

            while (buttonStop.isEnabled() == false) {
                try {
                    //System.out.println("Thread is going to wait...");
                    this.wait();
                    //System.out.println("Thread woke up...");

                } catch (InterruptedException ignore) {}
            }

            String ipadd = textIPaddress.getText();

            String toSend = new String("m\n");
            for (Trace2DXspan t : traces) {
                if (t != null) {
                    t.setXSpan(new Integer( textXspan.getText()));
                }
            }

            try {
                clientSocket = new Socket(ipadd, new Integer(textPort.getText()));
                os = new DataOutputStream(clientSocket.getOutputStream());
                br = new BufferedReader(
   new InputStreamReader(clientSocket.getInputStream()));

            } catch (UnknownHostException e) {
                System.err.println("Don't know about host: hostname");
                buttonStopActionPerformed(null);
            } catch (IOException e) {
                System.err.println("Failed creating sockets and streams");
                buttonStopActionPerformed(null);
            }

            try {
                bw = new BufferedWriter( new FileWriter(textLogFile.getText(),true));
            } catch (IOException ex) {
                System.err.println("Problem opening log file");
                buttonStopActionPerformed(null);
            }

            try {
                os.writeBytes("mi\n");
                bw.write( "\n" + br.readLine() + "\n");
                bw.write( br.readLine() );
            } catch (IOException ex) {
                System.err.println("Logging date and desciptive text failed");
                buttonStopActionPerformed(null);
            }


            while ( buttonStop.isEnabled() ) {
                if (clientSocket != null && os != null && br != null) {
                    try {
                        //System.out.print("Now writing: " + toSend );
                        os.writeBytes(toSend);

                        //System.out.println("Now reading...");
                        String responseLine = br.readLine();

                        if (responseLine != null ) {
                            //System.out.println(responseLine);

                            String[] responseValues = responseLine.split(",");

                            if (responseValues.length > 1) {
                                bw.newLine();
                                bw.write(responseLine);

                                Double xSinceEpoc = new Double(responseValues[1]);

                                if (xSinceEpoc < xFirst ) xFirst = xSinceEpoc;
                                xSinceEpoc = xSinceEpoc - xFirst;

                                Double y = new Double(0);
                                for (int i = 2; i < responseValues.length; i++) {
                                    y += new Double(responseValues[i])/1024;
                                    if (traces.get(i) != null) {
                                        traces.get(i).addPoint(xSinceEpoc, y);
                                    }

                                    // Ugly fix to get swapCached shown twice
                                    if (i == 9) {
                                        y += new Double(responseValues[9])/1024;
```

```
                                            if (traces.get(20) != null) {
                                                traces.get(20).addPoint(xSinceEpoc, y);
                                            }
                                        }
                                    }
                                }
                                try {

                                    Thread.sleep(400);
                                } catch (InterruptedException ignore) {}

                            } else {
                                //System.out.println("Null!! :-(");;
                            }
                        } catch (UnknownHostException e) {
                            System.err.println("Trying to connect to unknown host: " + e);
                        } catch (IOException e) {
                            System.err.println("IOException:  " + e);
                            break;
                        }
                    }
                }
                try {
                    os.writeBytes("q\n");
                    os.close();
                    br.close();
                    clientSocket.close();

                    bw.close();
                    bw = null;

                } catch (IOException e) {
                    System.err.println("IOException (closing):  " + e);
                }
            }
        }
    }
}
```

3.2 \info\monitorenter\gui\chart\traces\Trace2DXspan.java

```
/*
 *  Trace2DXspan is a modification of Trace2DLtd modified by Klaus Johansen.
 *  This modified version makes is "fix" the x-axis to a certain "span" width.
 *  Points with x < (newest x - span) are discarded.
 *
 *  Trace2DLtd is a RingBuffer- based fast implementation of a ITrace2D.
 *  Copyright (C) 2002  Achim Westermann, Achim.Westermann@gmx.de
 *
 *  This library is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU Lesser General Public
 *  License as published by the Free Software Foundation; either
 *  version 2.1 of the License, or (at your option) any later version.
 *
 *  This library is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 *  Lesser General Public License for more details.
 *
 *  You should have received a copy of the GNU Lesser General Public
 *  License along with this library; if not, write to the Free Software
 *  Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
 *
 *  If you modify or optimize the code in a useful way please let me know.
 *  Achim.Westermann@gmx.de
 */
package info.monitorenter.gui.chart.traces;

import info.monitorenter.gui.chart.Chart2D;
import info.monitorenter.gui.chart.ITrace2D;
import info.monitorenter.gui.chart.TracePoint2D;
import info.monitorenter.util.collections.IRingBuffer;
import info.monitorenter.util.collections.RingBufferArrayFast;
import java.util.Iterator;

/**
 * Additional to the Trace2DSimple the Trace2DLimited adds the following
 * functionality:
 * <p>
 * <ul>
 * <li>The amount of internal tracepoints is limited to the maxsize, passed to
 * the constructor.</li>
 * <li>If a new tracepoint is inserted and the maxsize has been reached, the
 * tracepoint residing for the longest time in this trace is thrown away.</li>
 * </UL>
 * Take this implementation to display frequently changing data (nonstatic, time -
 * dependant values). You will avoid a huge growing amount of tracepoints that
 * would increase the time for scaling and painting until system hangs or
 * java.lang.OutOfMemoryError is thrown.
 * <p>
 *
 * @author <a href='mailto:Achim.Westermann@gmx.de'>Achim Westermann </a>
 *
 * @version $Revision: 1.3 $
 */
public class Trace2DXspan extends ATrace2D implements ITrace2D {
    /**
     * Internal fast fifo buffer implentation based upon indexed access to an
     * array.
     */
    protected IRingBuffer m_buffer;

    private Integer xSpan;
    /**
     * Defcon of this stateless instance.
     */
    public Trace2DXspan() {
        this(new Integer(600), new Integer(30), Trace2DLtd.class.getName() + "-" +
      getInstanceCount());
    }

    public Trace2DXspan(final int maxsize, final int span) {
        this(maxsize, span , Trace2DLtd.class.getName() + "-" + getInstanceCount());
    }

    /**
     * Constructs an instance with a buffersize of maxsize and a default name.
     * <p>
     *
     * @param maxsize
     *          the buffer size for the maximum amount of points that will be
     *          shown.
     *
     * @param name
     *          the name that will be displayed for this trace.
     */
    public Trace2DXspan(final int maxsize, final int span, final String name) {
        this.m_buffer = new RingBufferArrayFast(maxsize);
        xSpan = new Integer(span);
        //xMax = new Integer(span);
        this.setName(name);
    }

    /**
     * @see ATrace2D#addPointInternal(info.monitorenter.gui.chart.TracePoint2D)
     */
    public boolean addPointInternal(final TracePoint2D p) {

        TracePoint2D removed = (TracePoint2D) this.m_buffer.add(p);

        if (removed == null) {
            TracePoint2D oldest = (TracePoint2D)this.m_buffer.getOldest();
            if ( oldest.getX()  < ((int)p.getX()) - xSpan ) {
                removed = (TracePoint2D)this.m_buffer.remove();
            } else {
                // no point was removed
                // use bound checks of calling addPoint
                return true;
            }
        }

        while (removed != null ) {

            double tmpx;
            double tmpy;

            tmpy = removed.getY();

            if (tmpy >= this.m_maxY) {
                tmpy = this.m_maxY;
                this.maxYSearch();
                this.firePropertyChange(PROPERTY_MAX_Y, new Double(tmpy),
       new Double(this.m_maxY));
            } else if (tmpy <= this.m_minY) {
                tmpy = this.m_minY;
                this.minYSearch();
                this.firePropertyChange(PROPERTY_MIN_Y, new Double(tmpy),
       new Double(this.m_minY));
            }
            // scale the new point, check for new bounds!
            this.firePointAdded(p);

            removed = null;
            TracePoint2D oldest = (TracePoint2D)this.m_buffer.getOldest();

            if ( oldest.getX()  < ((int)p.getX()) - xSpan ) {
                removed = (TracePoint2D)this.m_buffer.remove();
            }
            // scale the new point, check for new bounds!
        }
        return false;
    }

    /**
     * @see info.monitorenter.gui.chart.ITrace2D#getMaxSize()
     */
    public int getMaxSize() {
        return this.m_buffer.getBufferSize();
    }

    /**
     * Returns the acutal amount of points in this trace.
     * <p>
     * @return the acutal amount of points in this trace.
     * @see info.monitorenter.gui.chart.ITrace2D#getSize()
     */
    public int getSize() {
        return this.m_buffer.size();
    }

    /**
     * @see info.monitorenter.gui.chart.ITrace2D#isEmpty()
     */
    public boolean isEmpty() {
        return this.m_buffer.isEmpty();
    }

    /**
     * @see info.monitorenter.gui.chart.ITrace2D#iterator()
     */
    public Iterator iterator() {
        if (Chart2D.DEBUG_THREADING) {
            System.out.println("Trace2DXspan.iterator, 0 locks");
        }

        synchronized (this.m_renderer) {
            if (Chart2D.DEBUG_THREADING) {
                System.out.println("Trace2DXspan.iterator, 1 lock");
            }
            synchronized (this) {
                if (Chart2D.DEBUG_THREADING) {
                    System.out.println("Trace2DXspan.iterator, 2 locks");
                }
                return this.m_buffer.iteratorL2F();
            }
        }
    }

    /**
     * @see info.monitorenter.gui.chart.ITrace2D#removeAllPoints()
     */
    public void removeAllPointsInternal() {
        this.m_buffer.clear();
    }
    /**
     * <p>
     * Returns false always because internally a ringbuffer is used which does not
     * allow removing of values because that would break the contract of a
     * ringbuffer.
     * </p>
     * @param point
     *          the point to remove.
     *
     * @return false always because internally a ringbuffer is used which does not
     *         allow removing of values because that would break the contract of a
     *         ringbuffer.
     */
    protected boolean removePointInternal(final TracePoint2D point) {
        return false;
    }
    /**
     * Sets the maximum amount of points that may be displayed.
     * <p>
     *
     * Don't use this too often as decreases in size may cause expensive array
     * copy operations and new searches on all points for bound changes.
     * <p>
     *
     * TODO: Only search for bounds if size is smaller than before, debug and
     * test.
     *
     * @param amount
     *          the new maximum amount of points to show.
     */
    public final void setMaxSize(final int amount) {
        if (Chart2D.DEBUG_THREADING) {
            System.out.println("Trace2DXspan.setMaxSize, 0 locks");
        }

        synchronized (this.m_renderer) {
            if (Chart2D.DEBUG_THREADING) {
                System.out.println("Trace2DXspan.setMaxSize, 1 lock");
            }
            synchronized (this) {
                if (Chart2D.DEBUG_THREADING) {
                    System.out.println("Trace2DXspan.setMaxSize, 2 locks");
                }
                this.m_buffer.setBufferSize(amount);

                double xmin = this.m_minX;
                this.minXSearch();
                if (this.m_minX != xmin) {
                    this.firePropertyChange(PROPERTY_MIN_X, new Double(xmin),
      new Double(this.m_minX));
                }

                double xmax = this.m_maxX;
                this.maxXSearch();
                if (this.m_maxX != xmax) {
                    this.firePropertyChange(PROPERTY_MAX_X, new Double(xmax),
      new Double(this.m_maxX));
                }

                double ymax = this.m_maxY;
                this.maxYSearch();
                if (this.m_maxY != ymax) {
                    this.firePropertyChange(PROPERTY_MAX_Y, new Double(ymax),
      new Double(this.m_maxY));
                }

                double ymin = this.m_minY;
                this.minYSearch();
                if (this.m_minY != ymin) {
                    this.firePropertyChange(PROPERTY_MIN_Y, new Double(ymin),
      new Double(this.m_minY));
                }
            }
        }
    }

    /**
     * <p>
     * Method triggered by
     * <code>{@link TracePoint2D#setLocation(double, double)}</code>,
     * <code>{@link #addPoint(TracePoint2D)}</code> or
     * <code>{@link #removePoint(TracePoint2D)}</code>.
     * </p>
     * <p>
     * Bound checks are performed and property change events for the properties
     * <code>{@link ITrace2D#PROPERTY_MAX_X}</code>,
     * <code>{@link ITrace2D#PROPERTY_MIN_X}</code>,
     * <code>{@link ITrace2D#PROPERTY_MAX_Y}</code> and
     * <code>{@link ITrace2D#PROPERTY_MIN_Y}</code> are fired if the add bounds
     * have changed due to the modification of the point.
     * </p>
     *
     * @param changed
     *          the point that has been changed which may be a newly added point
     *          (from <code>{@link #addPoint(TracePoint2D)}</code>, a removed
     *          one or a modified one.
     * @param added
     *          if true the points values dominate old bounds, if false the bounds
     *          are rechecked against the removed points values.
     */
    public void firePointChanged(final TracePoint2D changed, final boolean added) {
        double tmpx = changed.getX();
        double tmpy = changed.getY();
        if (added) {
            if ( ((int)tmpx) > this.m_maxX ) {
                this.m_maxX = (int)tmpx;
                //this.expandMaxXErrorBarBounds();
                this.firePropertyChange(PROPERTY_MAX_X, null, new Double( this.m_maxX +1));
                this.m_minX = this.m_maxX-xSpan;
                this.firePropertyChange(PROPERTY_MIN_X, null, new Double(this.m_minX));
            }

            if (tmpy > this.m_maxY) {
                this.m_maxY = tmpy;
                //this.expandMaxYErrorBarBounds();
                this.firePropertyChange(PROPERTY_MAX_Y, null, new Double(this.m_maxY));
            } else if (tmpy < this.m_minY) {
                this.m_minY = tmpy;
                //this.expandMinYErrorBarBounds();
                this.firePropertyChange(PROPERTY_MIN_Y, null, new Double(this.m_minY));
            }
        } else {

            if (tmpy >= this.m_maxY) {
                tmpy = this.m_maxY;
                this.maxYSearch();
                this.firePropertyChange(PROPERTY_MAX_Y, new Double(tmpy),
       new Double(this.m_maxY));
            } else if (tmpy <= this.m_minY) {
                tmpy = this.m_minY;
                this.minYSearch();
                this.firePropertyChange(PROPERTY_MIN_Y, new Double(tmpy),
       new Double(this.m_minY));
            }
            if (this.getSize() == 0) {
                //this.m_firsttime = true;
            }
        }
    }

    public Integer getXSpan() {
        return xSpan;
    }

    public void setXSpan(Integer xSpan) {
        this.xSpan = xSpan;
    }

}
```